Report No. UIUCDCS-R-79-973 UILU-ENG 79 1724

CONTROL STRUCTURES FOR PARALLEL MACHINES

by

Frank Dagen Panzica

May 1979 NSF-OCA-MCS76-81686-000042

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-79-973

CONTROL STRUCTURES FOR PARALLEL MACHINES

by

Frank Dagen Panzica

May 1979

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF MCS76-81686 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, May 1979.

ACKNOWLEDGMENT

I wish to thank Duncan Lawrie for his help in catching 'off-by-one' errors and for his guidance throughout this project. I would also like to thank John Wawrzynek for his suggestions and for the many 'bells-and-whistles' added to his program, which made my work easier.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Recurrences have long been known to exist in scientific Fortran programs. Algorithms for solving these recurrences have also been developed. These algorithms are designed to solve a given recurrence in the least time. In most cases the number of processors, p, needed for the least time solution is found by counting the number of processor steps needed in the operation requiring the largest number of these processor steps. Several solutions have also been developed that assume a given number of processors are available and then adapt an algorithm to never require more than this number. Both of these solutions are fine in the theoretical sense, however, other considerations must be explored before these algorithms can be useful in speeding up the execution time of a program. The purpose of this thesis is to adapt these solutions to a generalized parallel machine.

The first problem encountered involves the scheduling of these algorithms. Memory conflicts, alignment network

usage, device overlap, as well as processor utilization must be considered. The algorithms mentioned above have been judged strictly by the number of processor steps needed to complete the algorithm. Here a processor step may be any arithmetic operation, such as addition, multiplication, or division, regardless of differences in complexity and execution time. This approach fails to consider other resources and the interconnection between resources used when solving the recurrence. These resources must be considered if a recurrence algorithm is to be used on an actual machine. Several other parameters that are fixed by a given architecture must also be considered and will be mentioned later in this introduction.

The second major problem with the algorithms as previously developed is the assumption of an unlimited number of available processors. In some cases this problem has been alleviated by the suggestion that if less processors exist, folding takes place. This assumption, in fact, seriously alters the times associated with each algorithm, and must be carefully considered when choosing the "best time."

This thesis discusses the algorithms currently considered to be "best." These algorithms will be redefined to specifically consider the resource scheduling problems

mentioned above. Machine architecture parameters such as sneak paths, register usage, and resource times will be considered. Finally, the advantages of representing recurrence algorithms as a combination of control structures will be discussed, and possible extension of this idea to micro-coded machine instructions will be mentioned.

This paper will be concerned with a generalized machine architecture. This means that no assumptions related to the number of memories, the alignment network(s), the number of processors, or the times associated with these resources will be made. This was done in order to study the effects of altering various architecure parameters. Because of this decision, the interaction between resources must be considered dynamic. Overlap between resources cannot be determined for a given algorithm without considering the times for each resource. This problem was solved by the development of a program FAPGEN [1]. Using this program, overlap can be dynamically calculated and algorithms can be studied on the higher level of resource usage without being concerned with the specific times of each resource.

While defining the theory of control structures, it became apparent that there exists the possibility of defining machine instructions to execute these control structures. This approach has numerous advantages in terms

of efficiency and simplicity when actually executing these algorithms on a parallel machine. More will be mentioned concerning machine instructions, throughout this thesis, whenever applicable.

Chapter 2 will discuss the fundamentals behind the use of control structures. Structures will be defined and a naming scheme will be explained. A canonical set of control structures will be diplayed to allow for flexibility in handling future algorithms. Next, a set of control structures occurring in the algorithms currently under study will be enumerated. Finally, future considerations for altering this set will be discussed.

Chapter 3 deals with the use of these control structures in recurrence algorithms. First a tutorial example will be presented to illustrate the general idea. Next, several algorithms will be discussed in detail.

Chapter 4 presents dynamic data concerning individual control structures. This data will be sampled with different machine architecture parameters and resource times. Both the number of repetitions of each control structure and the individual occurrences of each structure will be tabulated. Finally, structure usage will be considered on the basis of the likelihood of each algorithm

being used for a given set of recurrence bounds; i.e., algorithms requiring the least time will be considered.

CHAPTER 2

CONTROL STRUCTURES

2.1 Introduction to Control Structures

2.1.1 Definition of Control Structures

The term control structure will be used to refer to a block of resource requests that are in some way linked together. There is no required restriction on the number or type of resources that can be linked together into one structure, however, for the purposes of this thesis, memory access, alignment network usage and the processor are the only resources considered. Each node in a control structure represents one resource request for each processor/memory unit. Thus, if there are four processors available, a multiply will be done in all four processors simultaneously, but will be signified in a control structure by a single "*".

Control structures have been designed to display resource requests in a tree-like representation. The links between nodes of this tree imply a data or control dependency between the resources represented by the

respective nodes.

Each structure contains two or more resources and is intended to include as many related resource requests as possible. A related resource request is present when the data used, i.e. fetched, aligned, added, etc., by one resource is then used by the succeeding nodes linked to it. Upper bounds on the number of resource nodes included in any one structure are determined by the largest iteration independent group of resource requests (see section 2.2). For the algorithms studied so far (see App.B), thirteen is the maximum number of resource nodes included in any one structure.

The control structures mentioned here and throughout the rest of this thesis are closly related to potential hardware or firmware implementation as micro-coded machine instructions. With this in mind, a generalized computer organization is assumed. This organization is a circular arrangement of the resources: memory - alignment network - processor - alignment network. (Note: there may be either a single alignment or two alignment networks)

Provisions for sneak paths have been made, but sneak paths will be considered additional features of a system, not an inherent feature of the control structure. Thus, if

the alignment network can be bypassed, assuming the appropriate sneak path is available, the feature is noted, but the control structure will still contain the additional resource node (see section 2.1.2).

Throughout this thesis, control structures will be assumed to consist of the following resource nodes: memory access (fetch and store), alignment network(s) usage, and processor usage. The general format for control structures allows for inclusion of additional resource nodes if the need or desire exists. Generally, however, including resources other than those mentioned above is both awkward and potentially invalid.

Register usage will not be specifically mentioned, however conservative estimates on the number of registers available have been adhered to consistantly. In general, three registers will be assumed, two for inputs into a processor, and one for the output from a processor. Any case requiring an unusual number of registers will be noted.

2.1.2 Naming Conventions for Control Structures

One of the primary purposes behind the control structures expounded in this thesis is to facilitate the incorporation of new algorithms at the machine instruction

level. The control structures proposed here (see sections 2.3 and 2.4) have been found to be useful building blocks for describing and formalizing new recurrence algorithms. Developing a concise and unambiguous naming scheme was quickly found necessary.

First a definition of terms:

resource node - an elementary component of a control structure. Each node corresponds to one resource request (per processor).

F - memory fetch (uses the same resource specified by an S).

S - memory store (see F).

A - alignment network usage. Refers to both the input alignment network and the output network, if both exist. If an A-node is proceeded by an F-node or followed by a P-type node, it is assumed to refer

to the input alignment network (when one exists). If an A-node is proceeded by a P-type node or followed by an S-node then it is assumed to refer to the output alignment network.

P (or P-type) - resource node referring to a generalized processor step. Also used for null processor steps needed to allow for a generalized machine architecture.

+ - * / - addition, subtraction, multiplication and division. These are processor steps.

U- - unary minus (as opposed to subtraction).

? - nodes having this designation may be skipped if the appropriate

sneak path exists.

< - designates a backward data flow. The output of a node followed by this symbol is used as an input in a preceeding node.

The naming of control structures consists of combining the terms defined above with each term corresponding to a resource node with an optional '?' added to designate a possible sneak path, or a '<' added to designate a backward dependency. The beginning of each name includes a count of the number of resource nodes in the control structure. Thus a typical structure could be referred to as '7FAFA+A?S'. This name refers to a structure having seven nodes. Two pieces of data are fetched from the memory and aligned into a processor (FAFA). An addition takes place (+), followed by aligning the processor output and then storing the answer in memory (AS). The final align may be skipped if a sneak path, between the output of the processor and the memory, exists (A?). (Note: the node is included in the original count (7) whether or not the sneak path exists)

The use of '<' can best be explained with an example.

A simple control structure "2A+" can be used to add two numbers. By adding '<', we have "2A+<" which can be thought of as a chain sum. On the first iteration, both inputs to the processor come from registers. In every iteration after the first one, one input is the output of the last iteration.

The '<' appears immediately after the processor step that has as one of its inputs the output of a previous iteration. This notation is not completely unambigious, however some concessions must be made for managability. As with '?', '<' does not add to the resource count when naming the control structure.

For strictly linear structures no ambiguity exists. However, for complex structures a more detailed convention is needed. Whenever possible, an alignment is associated with the corresponding memory access. Thus 'FA' refers to fetching and aligning the same piece of date, likewise 'AS' refers to the movement of the same data. Also, all inputs to a processor must be mentioned before the processor step, thus 'FAFA+' is used, not 'FA+FA'. Furthermore, a '?' always immediately follows the node which may be skipped due to sneak paths. Finally, a generalized machine architecture is assumed, thus a reference such as 'FAS' is illegal, since this implies bypassing the processor. The operation of

fetching, aligning and then storing data must be represented by 'FAP?A?S' (see section 2.2.1), where the 'P' refers to a null processor step and both the 'P' and the second 'A' may be skipped if sneak paths exist.

2.2 Tradeoffs When Choosing Control Structures

A major consideration when presenting these control structures is their potential use as machine instructions. This results from the ability to decompose existing recurrence algorithms into a relatively small number of distinct control structures, with the potential speed-up of executing machine instructions as opposed to individual resource requests. The ease of expressing recurrence algorithms, as well as standard serial and vector operations, using control structures, and the ability to extend this set of control structures will also be considered (see section 2.3).

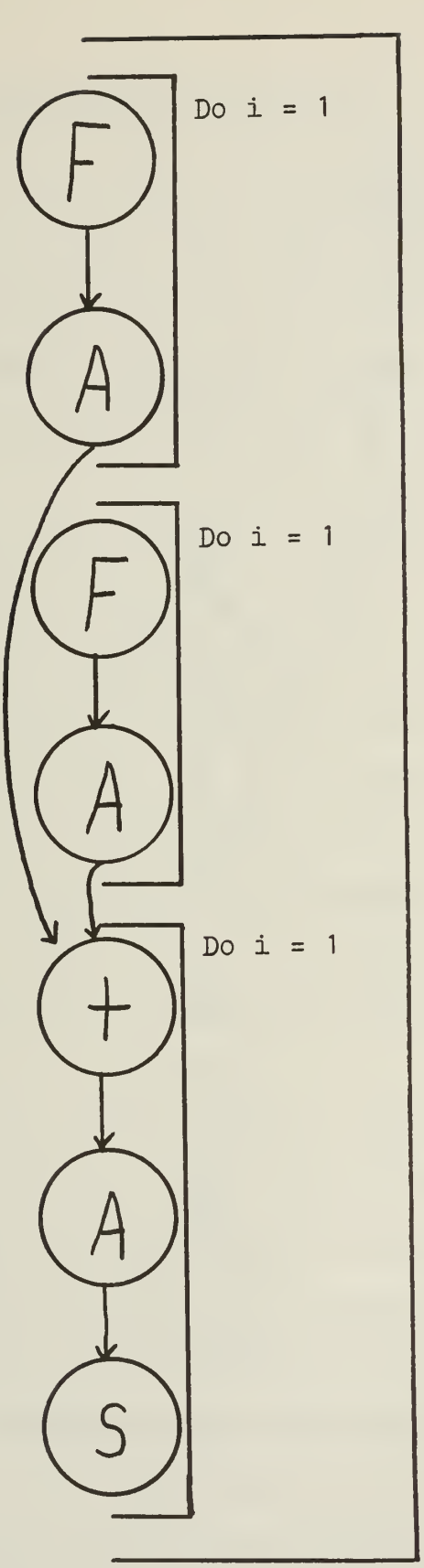
Control structures throughout this thesis are purposefully left as general as possible relative to any specific machine. Because of this decision, several pairs of control structures may exist that differ only in that some resources may be skipped due to sneak paths present in one structure, but absent in the other. For a specific machine these structures could be one machine instruction

with the appropriate resources optionally included or excluded from execution.

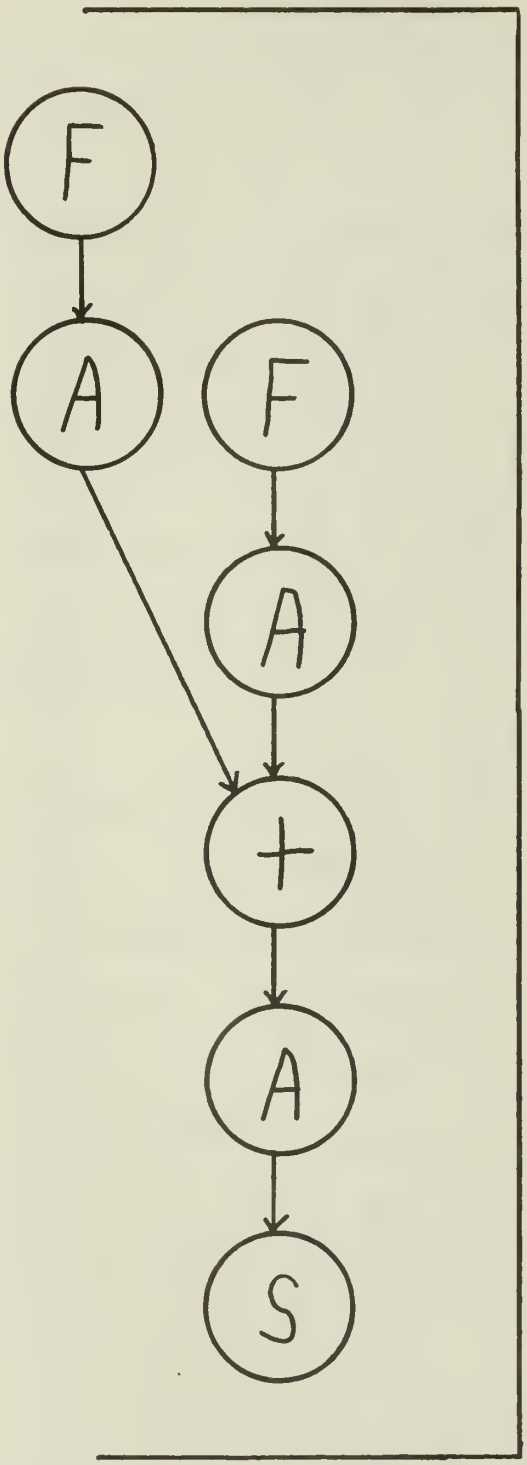
Another consideration concerns identical control structures with different processor steps; i.e., "7FAFA+AS" and "7FAFA*AS". Here, once again, the two structures will be considered distinct throughout this thesis, except for defining the canonical set. As machine instructions, however, these structures may be distinct, or may be combined (with appropriate selection control) depending on the choosen architecture.

Control structures become more efficient as they include more and more resource requests. Often there are several options available for representing a particular portion of an algorithm. Fig. 2-1 and fig. 2-2 show two possible representations of a vector add. In fig. 2-1 the vector add has been executed using three control structures, "FA", "FA", "+AS" and in fig. 2-2 only one control structure is needed. Two cases will illustrate the advantage of the approach used in fig. 2-2 as opposed to the approach used in fig. 2-1.

If a structure (machine instruction) is only executed once, little is saved by the second approach. However, even in this case there is some potential savings. If the vector



(fig. 2-1)



(fig. 2-2)

add is implemented as one machine instruction, the second F can be overlapped with the first align when defining the machine instruction. As three instructions, an outside controller would be needed to overlap the first two instructions. Thus, even in this simple case, using one instruction reduces complexity.

Even more savings are apparent when the vector add is executed more than once. As one instruction, overlap can be handled once (when defining the instruction) and thereafter the structure can be executed with maximal overlap. In the "7FAFA+AS" structure, the second F can be overlapped as mentioned above, but the first F can also be overlapped with the align out for all iterations after the first one. Thus, this approach has the advantage of maximizing overlap and requiring the fetching of only one machine instruction.

The approach illustrated in fig. 2-1 once again suffers in comparison. Here, three instructions must be fetched for each iteration (they could be stored in a buffer, but this of course requires a sufficiently large buffer to handle the general case). Overlap must be handled dynamically as mentioned above, and a looping structure must be set up.

Thus, there are many advantages to making control structures as inclusive as possible. This philosophy will

be followed throughout this thesis, unless specifically noted.

The process of determining how many distinct structures are present in a given algorithm is, however, open to modification. Currently, for reasons of flexibility and efficiency, control structures have been made as inclusive as possible. If two structues are executed sequentially, and each is executed only once, then these two structures are combined to form one new control structure. However, some structures formed by this approach are only used infrequently. An example of this is found in BINVERSE which finds the inverse of a lower-triangular, 2x2 matrix. In this algorithm, one structure is only executed if there are two processors available and the matrix is constant coefficient, another structure is executed in all other cases.

This specialization can artifically increase the number of unique control structures (machine instructions) that are needed to execute a given set of algorithms. The easiest way to circumvent this is to divide these specialized structures into a set of more widely used structures (a canonical set of structures will be defined in the next section). Chapter 4 will deal with the problem of deciding when such a decomposition is called for, by studying the

frequency of use and the number of repetitions of each structure, and thus deciding which structures are the most "useful".

2.3 A Canonical Set of Control Structures

2.3.1 Introduction

Through the efforts of representing recurrence algorithms as a combination of control structures, many such structures have been defined. In addition, structures have been added to represent vector operations. These sources have resulted in approximately fifty defined control structures. While no attempt at claiming this includes all possible structures is made, this set has been sufficient for describing at least two new algorithms [2] and [3] which have been proposed after much of the work for this thesis had been completed.

Section 2.3.2 will present a subset of the structures already defined which can then be used to form any of the other structures currently defined. This is not intended to replace these compound structures, but rather to present a canonical set for inclusion of future structure once the addition of new structures becomes impossible or extremely difficult; i.e., after the machine has been built.

As mentioned above, ideally structures should include as many resource requests as possible. This approach is fine when designing a machine or adding a new algorithm, however it is not always feasible to consider all possible combinations of resource requests. In order to demonstrate the flexibility of the structures defined so far, a canonical set will be defined in the next section.

The canonical set, as defined, is not intended to replace larger structures. The purpose of this set is to demonstrate the mathmatical completeness of this set of control structures as a cover of the desired structures. Since this set will be shown to exist, it is possible to define any new structure as a combination of the existing structures. The optimal case is to add a new structure whenever a new structure is needed, but this will not always be practical. The next best approach is to combine structures already defined. Because of the considerations demonstrated in figs. 2-1 and 2-2, it is always desirable to use as few (and as large) structures as possible.

For the purpose of defining a canonical set, we will restrict our discussion to control structures currently defined. The set F,A,P,S can, of course, generate any control structure (every control structure is defined as a combination of these resources). These are not currently

defined structures, and will not be considered as a canonical set (see the next section). The canonical set presented in the next section will be shown to handle all cases and thus replace the need for the individual resource requests mentioned above.

Justification of the canonical set will be discussed in section 2.3.3.

2.3.2 The Canonical Set

As mentioned earlier, the structures currently defined are designed for a generalized machine. For purposes of defining a canonical set, some distinctions between structures will be ignored. The first simplification that will be made is to ignore sneak paths.

Sneak paths result from the physical constraints placed on the hardware of a given machine. When the control structures are thought of as machine instructions, the presence or absence of a sneak path doesn't seriously alter that specific instruction. Instead, sneak path constraints can be thought of as externally controlled and not a fundamental characteristic of the instruction. This view results in considering the two structures "2FA" and "2FA?" as identical. If this is not desired, the canonical set

presented can be expanded to include all possible sneak paths by looking at the possible sneak paths present in the canonical set.

For the purpose of defining a canonical set, all process operations will be considered generalized processor steps and will be designated by a P. This eliminates any distinction between two structures, based only on a difference in processor steps. Once the canonical set has been defined, this simplification can be reevaluated. If a machine instruction is designed flexibly enough to handle choosing the specific processor step after the general instruction has been choosen, then this canonical set will be sufficient. If this is not the case, then the set can be expanded to include an example with each individual processor operation.

Processor steps modified by a "<" will be considered distinct. This relates to the different machine instructions needed to handle this case as opposed to the same control structure without "<".

Now we will discuss the set which can cover all defined structures, that is, a canonical set. First, the obvious elements of the canonical set are the smallest structures defined. Since these structures must be covered and aren't

comprised of smaller structures, they are included in the canonical set. Once the considerations of the above paragraphs have been applied, four unique two-element structures remain:

"FA"

"AS"

"AP"

"AP<"

"FA" and "AS" are easily recognized as standard, frequently used control structures. However, "AP" and "AP<" need some explanation. Here the P stands for any processor step (as mentioned above). Normally processor steps require two inputs, so in these structures, one input is a register that has to be aligned (no F so the data can't come from memory). The other input, for "2AP", can either come directly from a register, or be an element fetched from memory using "FA" which has already been mentioned. For "2AP<" the second input is the output of the processor step.

An obvious extension to the above are "FAP" and "FAP<". Here, one input to the processor is fixed as coming from memory, and the other input is optionally from memory or

from a register for "FAP", and from the processor, as mentioned above, for "FAP<". These structure also appear in the required set of control structures, so no new structure need be defined.

The only other unique three-element structure is "PAS". The output of the processor in this structure is stored in memory. No restrictions on the inputs to the processor are specified at this stage.

Four-element structures provide two additional members to the canonical set. The four-element structures are "FAPP" and "FAPP<". The second processor step in "FAPP" receives one input from the processor operation executed immediately before it, and receives the other input from an unspecified source. In "FAPP<", the other input to the second P is the output from the processor step during the previous iteration.

These structures both complete the canonical set and bring up an important question; i.e., if "FAP" and "FAPP" are included, what about "FAPPP", or "FAPPP<", etc.? This question will be discussed in section 2.3.3.

2.3.3 Sufficiency of the Canonical Set

First, the sufficiency of the canonical set can be easily shown for the structures defined so far. An example will illustrate this point.

A vector add is an elementary operation that can be described using a control structure. As one structure, this operation is simply "FAFA+AS". This can easily be shown to be composed of three elements of the above canonical set, that is, "FA", "FA", and "PAS". As it turns out, this structure can also be represented using "FA", "FAP", and "AS". Finally, the structure can be decomposed using an existing structure not contained in the canonical set as follows: "FA" and "FAPAS".

The above illustrates the process necessary to show the sufficiency of this canonical set for representing currently needed structures. Showing the decomposition of the other defined structures will be left for the reader.

Now to handle the problem presented in the last section, "Does the set cover all possible structures?" The answer is not really. First, we will show three solutions to this problem, and then the set already described will be defended. Following are the three alternative canonical

sets:

1)	2)	3)
"F"	"FA"	"FA"
"A"	"AS"	"AS"
"P"	"AP"	"AP"
"S"	"AP<"	"AP<"
	"P"	"FAP"
		"FAP<"
		"PAS"

Note: in sets 1 and 2, the structure "P" covers both the case where P has a backward data dependence, and when it doesn't.

Set 1 represents all resource requests, and thus this set must be a covering for the set of control structures under consideration. The disadvantage of this set is its inconsistancy with the intention behind control structures as mentioned throughout this thesis. Control structures are important for the efficiency associated with combining resource requests. Since none of these elements are currently defined structures, making this a canonical set

would require adding these elementary structures. This approach detracts from the desired results, and will thus not be considered further.

Set 2 has the advantage of solving the problem of "FAPP" as well as being flexible in allowing for unforseen structures. The disadvantage of this set is its failure to fully utilize the benefits of the control structure concept, since the fourth element of the set, P, reverts back to the idea that each resource request must be handled separately. Also, P, is not currenty used as an individual structure in any of the algorithms under study, thus it must be defined strictly for its use in the canonical set.

Set 3 is the canonical set previously defined, minus the eigth element, "FAPP". "FAPP" has been dropped to avoid the question of if "FAPP", why not "FAPPP"? A special case must be defined to represent "FAPP" which we know is a "needed" structure. Refering back to section 2.3.1, it will be remembered that we are ignoring sneak paths. If instead, we consider a sneak path to exist for "AP" (remember "A?P") then "FAPP" can consist of a combination of the two structures "FAP" an "AP", with the sneak path always on in "AP". The same applies for "FAPP<", etc.

The above approaches "solve" the problem, but fail to

fully appreciate the control structures as presented here. "FAPP" is an extremely important structure (see section 2.4) and should not require a special case to define it. Also, the entire flavor of the high level nature of control structures is jeopardized if "P" is allowed to be a structure. A compromise seems justified.

For all structures currently defined, the following set forms a minimal covering.

4) "FA"

"AS"

"AP"

"AP<"

"FAP"

"PAS"

"FAPP"

"FAPP<"

"FAPPP" is not included because it doesn't come up. The above set has withstood the addition of two new agorithms and several algorithm modifications and is still

valid. There appears to be no logical need for structures like "FAPPP". Finally, if such a structure is needed in the future, two solutions exist. One, use an arguement similar to the one used in critiqueing set 3, or two, define a new structure and add it to the canonical set. The first approach preserves the integrity of the given canonical set and can be justified if the new structure is infrequently used, while the second solution is justified if the new structure is used often. If the structure does appear often it is better for it to exist as a single entity, as opposed to a collection of structures, in order to best take advantage of the efficiency of the control structure concept.

More remarks will be presented on the issues concerning the last thought in the next section.

2.4 Additional Control Structures

As mentioned in the above section, all existing control structures can be formed from a combination of the elements contained in the canonical set. While this is necessary for generality, this idea does not use the fundamental idea behind the control structure concept. This idea is that there are only a limited number of unique control structures found in practice - no matter how complex the application.

These structures are bounded by iteration counts and loop limits, not by artifically selected restrictions. The small number of control structures needed reflects the nature of the algorithms and does not reflect the architectural design of a given computer. These facts combine to suggest the expedience of executing some given set of control structures as machine instructions. The previous section presented a set of minimal control structures and demonstrated their universality. This section will present some more complex structures that occur in several algorithm descriptions. Chapter 4 will look at the frequency of use for these control structures as further motivation for extending the idea of control structures to the definition of machine instructions.

Several structures seem to play an important role in defining new algorithms. This section will look at two major categories of structures. Using the simplifications mentioned in the last section (no sneak paths and the use of a generalized processor step, P) the two structures discussed here will be "FAFAPAS" and "FAFAPFAPAS".

First, we will discuss "FAFAPAS". This basic structure occurs frequently in various related forms - "FAP", "FAFAP" and "FAPAS", as well as the already mentioned "FAFAPAS". The only difference between these structures is the memory

usage. Some of the inputs to the processor come directly from memory (every occurence of "FA") and some from registers previously assigned the desired value. Also, the result of the processor step is either routed to memory ("AS") or temporarily stored in a register.

The interest in this structure relates to its ability to represent vector operations. As mentioned before, a vector add (for example) is represented in control structures as "FAFA+AS". Structures of this type, that is, diads, can thus be represented using a single control structure. Diads occur frequently in programs and can be handled by control structures or machine instructions having two inputs. Triads, machine instructions requiring three inputs, also occur frequently in recurrence algorithms and matrix operations, and will be discussed next.

The other general structure worthy of note is "FAFAPFAPAS" and its simplified forms - "FAPP", "FAFAPP", and "FAPFAPAS". Once again, the difference between these structures relates to memory access.

An interesting feature concerning these structures involves the processor steps involved. In all cases, the first P is a multiply and the second processor step is either an addition or a subtraction. This fact denotes a

specific function occuring whenever a member of this group is present. This structure represents the row-column multiplication present in a matrix-vector or a matrix-matrix multiply. The presence of this structure is thus closely related to matrix operations. Even more important, however, is the fact that an algorithm containing matrix operations can easily be expressed at the control structure/machine instruction level by including this structure.

The third class of additional structures are all those which include nodes having backward dependencies, i.e., those with "<". These structures are related to the two classes above, except for the data dependency. Generally, these structures occur whenever a structure is executed more than once, and ends with a processor step, as opposed to a store. That is, "FAP" is fine if it is executed only once, but if it is executed more often than that, the output has to go some place useful. The structure "FAP<" then becomes the obvious solution.

One more note concerning the existing control structures before we go on to discuss their use in recurrence algorithms. This comment concerns a class of structures not previously mentioned. This class includes all structures explicitly showing more than two inputs into a processor; i.e., "FAFAFAFAP". These structures assign

data to only half of the array of processor with any one memory access. Thus, the first "FA" supplies data to half the processors (usually either the left or right half of the processors, or all even processors, etc.). The next "FA" supplies data to the same location in the the other half of the processors, etc.

Now we will move on and describe how the control structures defined here can be used to facilitate the definition and execution of recurrence algorithms. Chapter 3 will present some algorithms, and Chapter 4 will look at the frequency of use of these structures in the algorithms when they are being executed.

Appendix A gives a pictorial description of all defined structures.

CHAPTER 3

RECURRENCE ALGORITHMS

3.1 Tutorial Example(MVMULT)

3.1.1 Introduction to MVMULT

For the algorithm discussed here (MVMULT) and all others mentioned in this thesis, some prior familiarity with the algorithm is assumed. The task of representing an algorithm with a limited number of control structures (as proposed in this thesis) must require at least this background proficiency. Continuing beyond this basic understanding, this section will attempt to explain, step by step, a procedure for representing an algorithm using control structures.

Matrix-vector multiplication (MVMULT) was chosen as a tutorial example for two primary reasons. One, matrix-vector multiplication is a widely understood algorithm with which the reader is assumed to be familiar. Also, matrix-vector multiplication is an extensively used operation in the parallel recurrence solvers currently under consideration. This fact reflects the desirability of

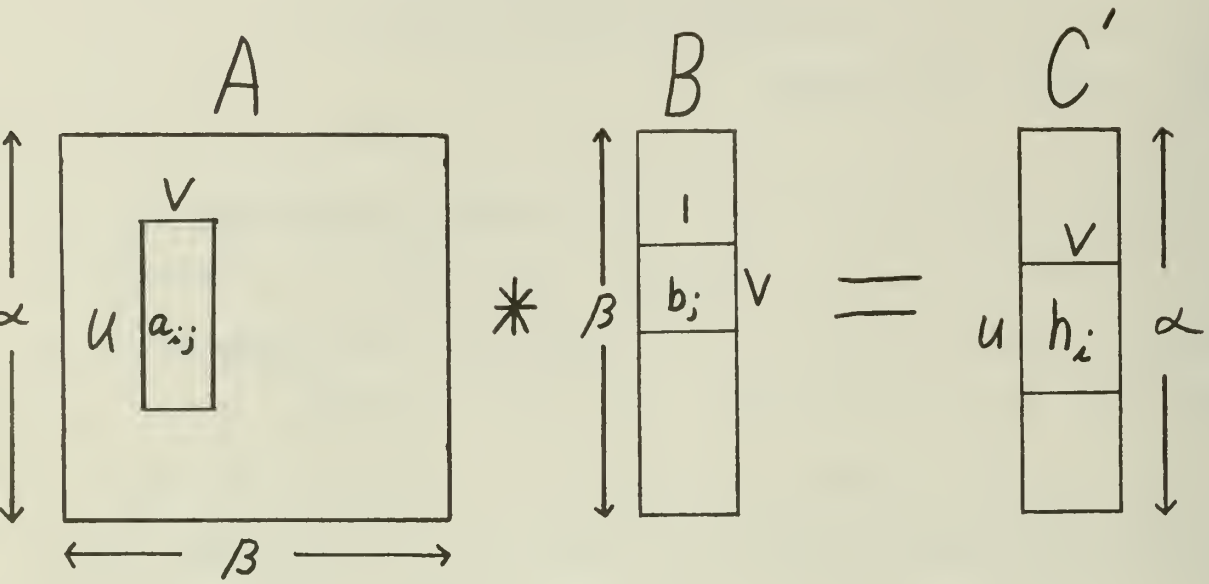
expressing algorithms in matrix notation and should be an important consideration in future algorithms. (Note: see CCLGL for an algorithm almost entirely composed of matrix-vector multiplies)

The matrix-vector algorithm used here is designed to utilize the available processors. The algorithm is executed by partitioning the matrix and vector as in fig. 3-1, so that each partition contains P (or less) elements.

3.1.2 Defining Parameters

Several factors must be taken into consideration when preparing the formalization of an algorithm. Machine architecture parameters can be known values or left as variables. The first class of parameters to be considered are the high level parameters of the algorithm - in this case, p (the number of processors available), Alpha (one dimension of the matrix), and Beta (the other dimension of the matrix). For a recurrence algorithm, the important parameters become p, N (the size of the recurrence), and M (the bandwidth).

Once these bounds have been established, a high level description of the algorithm can be completed. For the example of MVMULT, the following is representative of a



Here,

$$h_i = \text{rowsum} (a_{ij} \times b_j \; \forall \; j)$$

a_{ij}, h_i are p/⌈p/Alpha⌉ by ⌊p/Alpha⌋ Beta

b_j is ⌊p/Alpha⌋ Beta

(fig. 3-1)

procedure to obtain C in the matrix equation A * B = C, where A is an alpha by beta matrix, B is one by beta, and C is one by alpha.

In fig. 3-2, $\lceil\ \rceil$ denotes the ceiling function. One other function used in this description needs some further explanation. This function will be referred to as a bounding function, BND. The purpose of BND is to place an upper and lower limit on the value contained within the $\lfloor\ \rfloor$. The maximum value of the function will be the value of the parameter outside the $\lfloor\ \rfloor$ (Beta in fig. 3-2). The lower bound will always be assumed to be one.

This function becomes necessary when trying to place limits on the execution of various stages of an algorithm. Some steps must be executed at least once, thus the lower bound of one. The occasion also arises for the use of this function as a divisor. This function is then used to insure that a given step is never executed more than beta number of times.

Finally, a_{ij} and b_j are the standard matrix representations for the elements of the A and B matrices, respectively. Rowsum is the summation of subproducts of the multiplication step. This rowsum will be done in log-time.

```
Do i = 1, ⌈Alpha/p⌉
     temp_i = 0
     Do j = 1, ⌈Beta/⌊p/Alpha⌋ Beta⌉ (1)
          temp_i = temp_i + a_ij * b_j
     end
     c_i = rowsum (temp_i)
end
```

[1] This is the BND function.

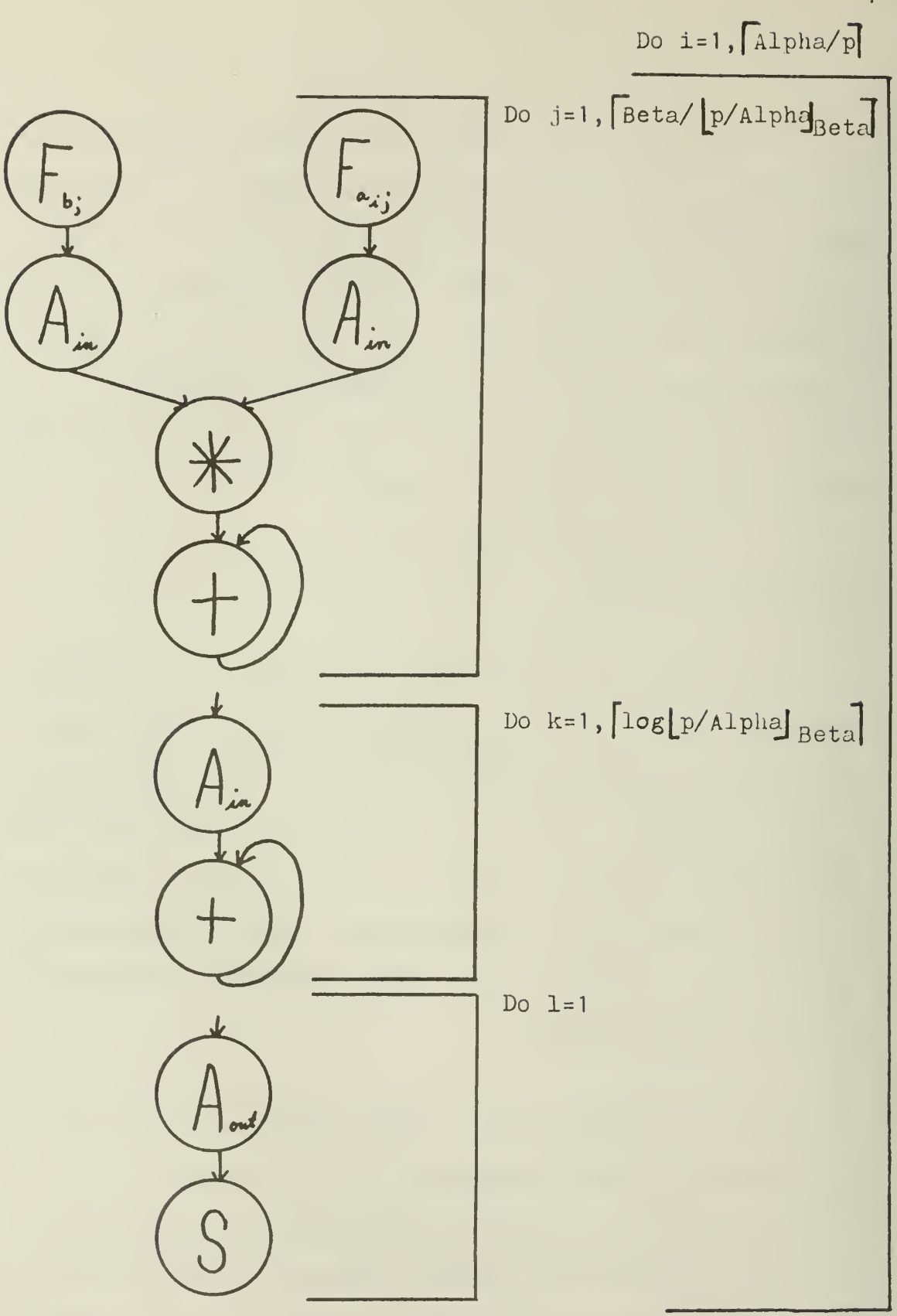
(fig. 3-2)

Now that the first level of formalization has been completed, the next set of parameters must be decided upon. These parameters are the resources to be considered (and used) when implementing the algorithm. In general, it will be assumed that a memory unit, an alignment-in and an alignment-out network, and a processor will be the enumerated resources. If additional (or fewer) resources are desired the procedure is similar.

3.1.3 The Algorithm

It is helpful to represent the algorithm pictorially (see fig. 3-3). Here the loops presented in (fig. 3-2) are expanded to illustrate resource usage. The single statement $temp_i = temp_i + a_{ij} * b_j$ has been rewritten to show the fetches (memory accesses) of the a_{ij} and b_j elements, their alignment into the processors, their multiplication and eventual summation. The other statements in fig. 3-2 have also been expanded.

This is the beginning of the representation of the algorithm as control structures. The standard transformation is to take the resources enclosed by a Do Loop and represent the resource allocation within that loop as a control structure. Nodes outside a loop, or inside a loop but at the same level as another loop (the final align


Do i=1,⌈Alpha/p⌉
Do j=1,⌈Beta/⌊p/Alpha⌋Beta⌉
F bj
F aij
A in
A in
*
+
Do k=1,⌈log⌊p/Alpha⌋ Beta⌉
A in
+
Do l=1
A out
S

(fig. 3-3)

and store, for example) can also be combined into control structures. Thus fig. 3-3 has three separate structures "6FAFA*+<", "2A+<", and "2AS" (see section 2.1 for naming conventions).

The next step is to determine if these structures are already defined (see App. A). If they are, then this stage is done. If not, there are three options. One, simulate the structures not found by combining a subset of the structures already defined. Since this set of structures includes a canonical set (see section 2.3) this will always be possible. The second option is to try and redo fig. 3-3 so that the structures currently defined can be used to implement the algorithm. The third option is to define a new structure.

Generally, all the structures will be present, as they are in this example. One side benefit of this fact is that if a structure or combination of structures appear in more than one algorithm, it indicates that these algorithms can be explained using similar notation. This can be useful in expressing different algorithms in the same notation, or in recognizing matrix operations (such as matrix-vector multiply), even if these operations are not explicitly mentioned at the level of fig. 3-2. This knowledge can greatly simplify an algorithm by suggesting a restatement of

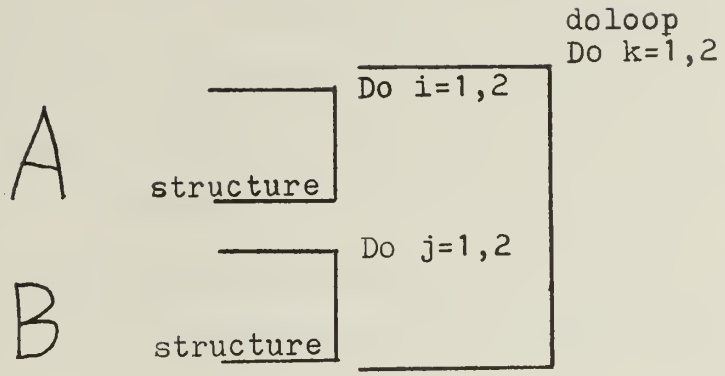
the algorithm in matrix notation.

Once structures have been determined, they must be linked together. This takes the form of adding "DOLOOPS" (see [1]), enumerating dependencies between structures, determining the number of times a given structure should be executed, and adding general control features (conditionals, etc.).

Optimally, a given structure should be executed with a repetition factor greater than one. This increases overlap and takes advantage of the similarity between the control structures and machine instructions (see chapter 4). In this algorithm, "2AS" is an example of a structure that can only be executed once; such structures are then handled differently. The "DOLOOP" construction is reserved for grouping structures that are repeated as a group. The difference between these two repetition factors is illustrated in fig. 3-4.

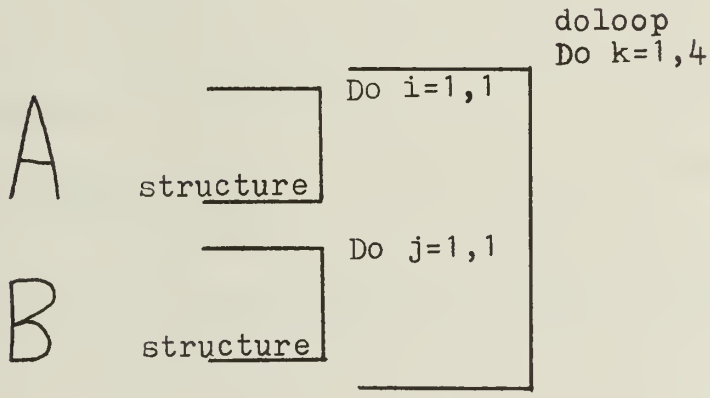
3.1.4 Completing the Algorithm

Dependency between nodes in sequential structures should be considered next. For clarity, some dependency should be specifically mentioned between every pair of structures, even if it is only a trivial dependency.



produces AABBAABB

- while -



produces ABABABAB

(fig. 3-4)

Trivially dependent here means a dependency between resources that must be handled sequentially for other reasons. An example of a trivial dependency can be found in the MVMULT algorithm where the align in "2AS" is dependent on the "+" in "6FAFA*+<", and trivially dependent on the "*", since the "+" must be executed after the "*" for other reasons; i.e., the nature of the structure.

The final set of parameters to be considered are resource times. For a given machine architecture, this presents no problem as these times are fixed. For a general computer architecture these times are considered variables that must be decided upon. For the sake of the algorithm presentation discussed here, these values are transparent and will be ignored until chapter 4. These values are, however, very important for determining overlap and should be considered when judging the feasibility of a given algorithm.

Before presenting the finished algorithm, one more point should be considered. As mentioned before, algorithms are most efficient when all operations can be expressed at the structure level. Upon checking the bounds for the Do Loops in MVMULT, it becomes apparent that if the Do Loop on i (i-loop) is done more than once ($\lceil alpha/p \rceil > 1$) then the j-loop ("6FAFA*+<") is only done once and the k-loop is

never done. By recombining, the algorithm can be written as one structure;G i.e., "8FAFA*+AS". The structure, for this case, can then be done with an iteration of alpha/p and with all "DOLOOPS" eliminated.

The complete algorithm appears in fig. 3-5. DOLOOP, MAP, IMAP, ENDO, and DEFINE are functions defined in [1]. These functions take structures and determine execution times. CEILQ is a function that returns the ceiling of the quotient. BND has already been defined, and CLOG2 returns the ceiling of the log base two of the expression.

3.2 Multiple Column Sweeping

3.2.1 Introduction

Column Sweeping is a classical algorithm for solving recurrences. This is due in part to its simplicity; i.e., solve for x then substitute and solve for x , etc. This method takes n-1 steps and requires m-1 processors at each step. For P = O(n) and with a large bandwidth, m, this algorithm is quite efficient. However, since most practical cases involve small values of m, some modification of the algorithm will be needed to handle these cases effectively.

```
MVMULT:  Proc (Alpha, Beta, P, Tptr) ;

     Call DOLOOP (1) ;
          bound = CEIL  (alpha/p) ;
          temp = bound * beta ;
          if bound > 1
          then call MAP ('8FAFA*+AS', temp) ;
          else do ;
               bound2 = CEIL  (beta/ BND ((p/alpha),beta) ;
               call MAP ('6FAFA*+<',bound2) ;
               bound3 = CLOG2 (BND ((p/alpha), beta)) ;
               if bound3 > 0
               then do ;
                    call DEPEND (6,1) ;
                    call MAP ('2A+<', bound3) ;
                    call DEPEND (2,1) ;
               end ;
               else call DEPEND (6,1) ;
               call IMAP ('2AS') ;
          end ;
     call ENDO (tptr) ;
end MVMULT ;
```

(fig. 3-5)

3.2.2 Defining the Algorithm

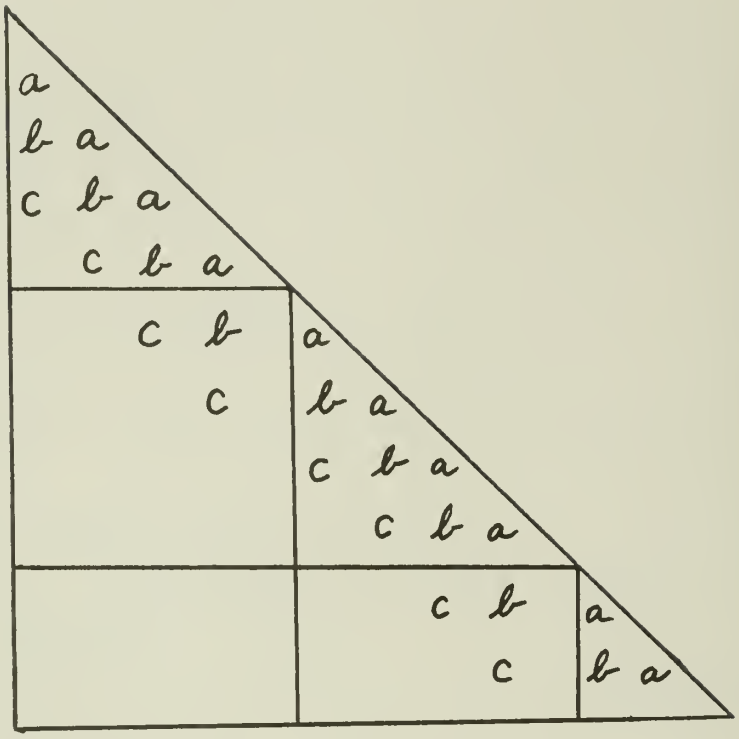
Start by looking at the banded matrix (recurrence) as subdivided into smaller matrices, some lower triangular, the A's, and some upper triangular, the B's (see fig. 3-6). Now, instead of just knowing the value of x_i, as is generally assumed in the classical case, the matrix A_i is taken as the known value, then the Column Sweeping algorithm can be written in the following matrix notation.

(1) $$x_i = f_i + B_i x_{i-1} + A_i x_i$$

(2) $$A_i x_i = f_i + B_i x_{i-1}$$

(3) $$x_i = A_i^{-1} (f_i + B_i x_{i-1})$$

Assuming the blocksize, Q, is chosen to maximize the use of the processors available (since the final stage requires multipling a QxQ matrix, P = O(Q*Q) would be reasonable, see the next section), then this calculation will require fewer steps than the classical version. This could considerably speed up the calculation. Unfortunately, equation (3) requires knowing A_i^{-1} for every A. This, in general, would require time consuming calculations and defeat the purpose of the modification. However, two special cases do exist.



(fig. 3-6)

If the inverse of A_i is known for every i then the problem has been solved. One such case occurs with constant coefficient recurrences. Here all the A's are identical, thus once the initial A_i is found, all of the inverses of A are known.

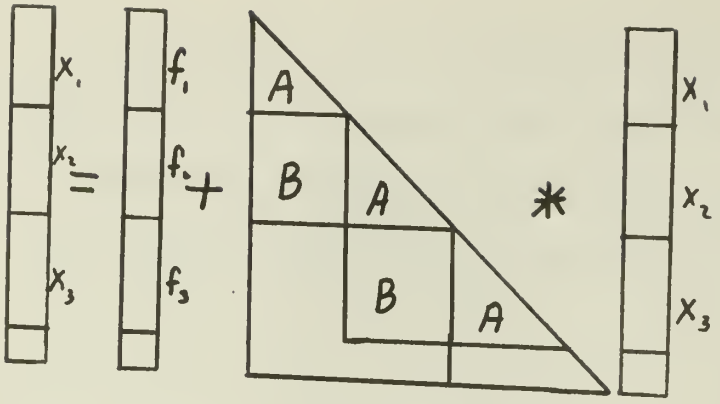
The other special case is when A is a 2x2 matrix. Here the inverse of A can be found with minimal effort and thus does not seriously invalidate the modified algorithm.

Efforts have been made recently toward handling the variable coefficient case for blocks larger than 2x2. The analysis of this case is similar to the methods described, and thus will not be mentioned further here.

In the next section the implementation of the CCLGL algorithm will be discussed.

3.2.3 Implementing the Algorithm

For constant coefficient matrices the algorithm is very straightforward (see fig. 3-7). First assume the inverse of A has been found by some means. Then the algorithm becomes simply a call to MVMULT for the calculation $B*x_{i-1}$, a call to VVADD (vector-vector add) for $f_i + B*x_{i-1}$ and a call to MVMULT for $A_i^{-1}*(f_i + Bx_{i-1})$. Since B is upper triangular (see

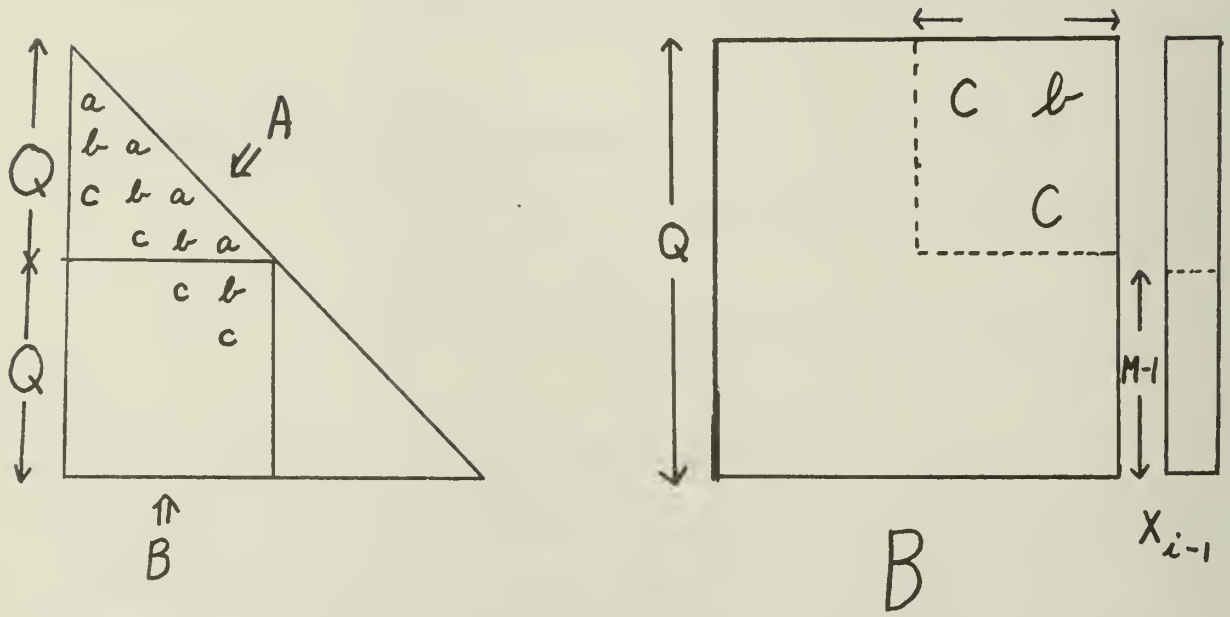


$$\text{set} \quad x_i = f_i + B*x_{i-1} + A*x_i \Rightarrow$$

$$A*x_i = f_i + B*x_{i-1}$$

$$x_i = A^{-1} * (f_i + B*x_{i-1})$$

(fig. 3-7)

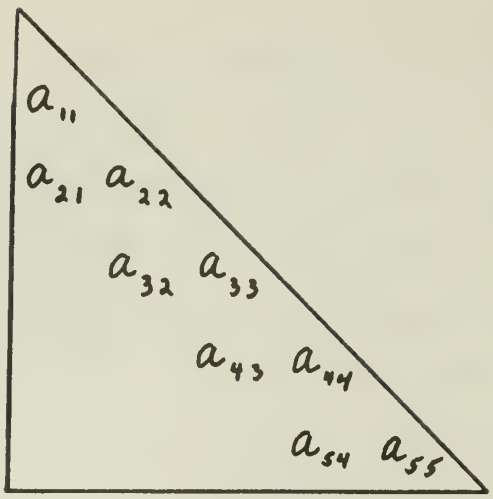


(fig. 3-8)

fig. 3-8) the first matrix-vector product will be with a (mxm) matrix and a (mx1) vector. The two (Qx1) vectors will then be added and finally a (QxQ) matrix will be multiplied by a (Qx1) vector. Since the size of the recurrence, n, need not be an even multiple of Q, a final stage will be needed to handle the last few rows. Thus the algorithm will sweep FLOOR (n/Q) times followed by a final sweep to pickup the final n - FLOOR (n/Q)*Q rows.

For the non-Toeplitz case (see fig. 3-9),where Q = 2, one additional stage is added. This stage calculates the inverse of a 2x2 matrix. Since this structure hasn't been mentioned yet, some details are in order. To find the inverse of a lower triangular 2x2 matrix one needs the following results: 1/a, 1/d and the product -c/ad. With 3 or more processors one can simultaneously divide 1/a, 1/d, and c/a. Follow this with multiplying c/a * 1/d. Negate the product and the inverse has been found (see fig. 3-10). If a = d (constant coefficient) then 2 processors will suffice. Otherwise, or if p=1, additional division and multiplication steps will have to be added.

The classical Column Sweep algorithm will not be discussed here, but note that if p < m, folding must take place (Note: folding throughout this thesis refers to the steps required to execute a segment of code requiring P



(fig. 3-9)

$$A = \begin{bmatrix} a & \phi \\ c & d \end{bmatrix} \quad A^{-1} = \begin{bmatrix} \frac{1}{a} & \phi \\ \frac{-c}{ad} & \frac{1}{d} \end{bmatrix}$$

(fig. 3-10)

processors, when only p processors are availabe and p < P. Folding normally will consist of executing a part of an algorithm CEIL (P/p)-times in order to execute the required number or steps. In some cases, this folding step might also require additional storage to hold temporary variables normally saved in registers).

3.3 The CCLGL Algorithm

3.3.1 Introduction

Recurrence algorithms historically have been designed to solve full or banded recurrences. In practice, recurrences found in FORTRAN are often constant coefficient recurrences. Based on the obvious assumption that constant coefficient systems are less complex, it is justified to assume that algorithms designed specifically for constant coefficient systems will be less involved and quicker in execution time.

CCLGL is a modification of the algorithm first proposed in [4]. The algorithm, as modified, is applicable for any combination of the bounds on n (recurrence size), m (recurrence bandwidth), and p (number of processors), although best performance is obtained with P > n. The algorithm differs from the original in two major respects.

One, the storage allocation is less complex, and two, after an appropriate time (see below) the algorithm reverts to multiple column sweeping (see section 3.2). The first change results in fewer memory accesses with a corresponding time savings as well as a decrease in the number of processors needed. The second change is necessitated by noticing that for some values of n, m, and p, the CCLGL algorithm will take more time than the generalized version appearing in [5]. By switching to multiple column sweeping, the timings for this algorithm are consistantly better over a wider variety of parameters.

The next sections give a more detailed analysis of this algorithm and an organization on the control structure level will be presented.

3.3.2 Defining CCLGL

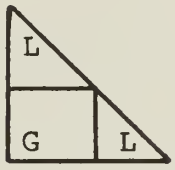
The following is an algorithm to solve constant coefficient recurrences with a bandwidth, m >= 2 (including the main diagonal). The algorithm will use P > O(n) and will be refered to as CCLGL.

The recurrence we wish to solve can be written as:

(1) $$Tx = f \Rightarrow x = T^{-1}f$$

Here T is a lower triangular, Toeplitz (constant coefficient), banded matrix (nxn). x and f are vectors (nx1).

Since T is constant coefficient, it can be decomposed as follows:

T = 

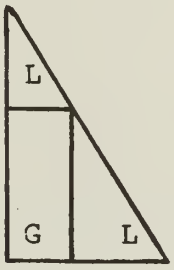
Two theorems will be useful in solving (1).

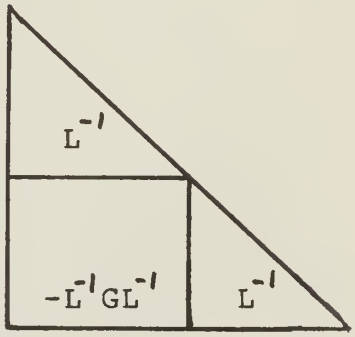
Theorem 1:

If T is Toeplitz, then T^{-1} is Toeplitz.

This is an immediate consequence of T being lower triangular and constant coefficient.

Theorem 2:

If T = 

, then T = 

Here, the L triangles are obvious (remember, L is

Toeplitz). The $-L^{-1}GL^{-1}$ section results from considering the matrix operations involved.

Remember, we are solving $x = T^{-1}f$. Since T^{-1} is Toeplitz (constant coefficient) we only need to solve for the first column of T . The system can thus be decomposed as in fig. 3-11.

The size of L doubles at each iteration; i.e.,

$$L_1^{-1} = 2x2 \; ; \; L_2^{-1} = 4x4 \; ; \; L_3^{-1} = 8x8 \text{ etc.}$$

The following equations result:

$$x_1 = L_1^{-1}f_1$$

$$x_2' = L_1^{-1}GL_1^{-1}f_1 + L_1^{-1}f_2$$

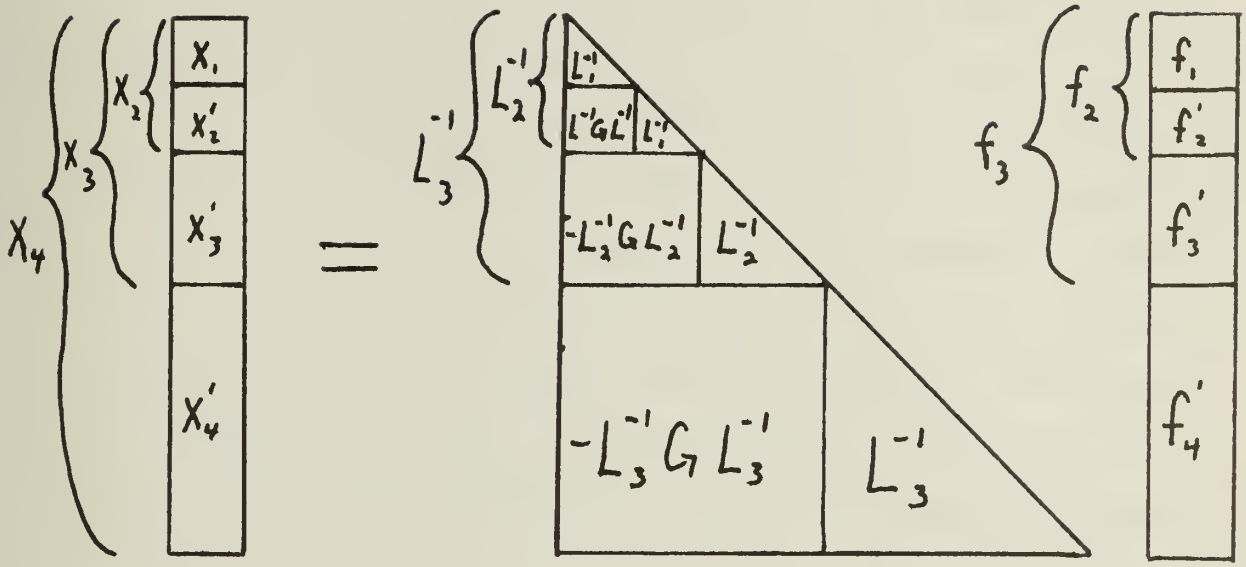
$$x_3' = L_3^{-1}f_3 = -L_2^{-1}G_2L_2^{-1}f_2 + L_2^{-1}f_3'$$

etc.

The general form becomes: ($i > 1$ and $n = 2$)

$$x_i' = -L_{i-1}^{-1} \; G_{i-1} \; L_{i-1}^{-1} \; f_{i-1} + L_{i-1}^{-1} \; f_i' = L_i^{-1} \; f_i'$$

(nxl) (nxn) (nxn) (nxn) (nxl) (nxn) (nxl) (nxn) (nxl)


X4
X3
X2
X1
X2'
X3'
X4'
=
L3^-1
L2^-1
L1^-1
L1^-1 G L1^-1
L1^-1
-L2^-1 G L2^-1
L2^-1
-L3^-1 G L3^-1
L3^-1
f3
f2
f1
f2'
f3'
f4'

(fig. 3-11)

Since $x'_{i-1} = L^{-1}_{i-1} f_i$,

$$x'_i = -L^{-1}_{i-1} G_{i-1} x_{i-1} + L^{-1}_{i-1} f'_i$$

Thus the solution resolves to 3 matrix-vector multiplies and one vector-vector add. Note: the temptation exists to simultaneously multiply $G_{i-1} x_{i-1}$ and $L^{-1}_{i-1} f'_i$, however, this requires twice as many processors and most importantly, creates potential memory conflicts by requiring access to 4 arrays simultaneously. Thus, this slight (potential) speed-up would not be constructive.

Finding L requires a simple 2x2 matrix inversion (even easier since L is Toeplitz). Also, since L is Toeplitz, we only need the first column and thus L can be considered an (nx1) vector in the following calculations. The algorithm reduces to:

I Find the inverse of L

II Do i = 2, $\log_2 n$

$$L^{-1}_i e_1 = \begin{bmatrix} L^{-1}_{i-1} \\ -L_{i-1} G_{i-1} L_{i-1} \end{bmatrix} L \quad * e_1 = \begin{bmatrix} L^{-1}_{i-1} e \\ -L_{i-1} G_{i-1} L_{i-1} e_1 \end{bmatrix}$$

III $x_{\log n} = L_{\log n} f_{\log n}$

The size of $L_{i-1}^{-1} G_{i-1} L_{i-1}^{-1} e_1$ is 2^{i-1}. Since T has a bandwidth of m, eventually G_{i-1} becomes upper triangular. The premultiplication by L_{i-1}^{-1} can also be reduced, in this case to a $(2^{i-1} \times m)$ matrix times a (mx1) vector. Thus the algorithm requires the following matrix-vector products:

$$(mxm) * (mx1)$$

$$(2^{i-1} xm) * (mx1)$$

$$(2^{i-1} x 2^{i-1}) * (2^{i-1} x 1)$$

The most processor intensive step is the final calculation of $L^{-1}(GL^{-1})$ which requires n*m/2 processors. The final stage (III) requires P = O(n*n) for optimal time, but of course can be folded.

For large n, the parallel time can exceed the serial time, thus a modification is needed. By looking at the T matrix as decomposed into two kinds of regions (A and B) (see fig. 3-6), the algorithm can be rewritten as:

Stage I

a) Solve $x_1 = f_1 + A_1 x_1$ (using the above algorithm)

b) Form $x_1 = L^{-1} f_1$

Stage II

Do i = 2, CEIL (n/Q)

$$x_i = f_i + Bx_{i-1} + Ax_i \quad \Rightarrow$$

$$x_i = L^{-1}(f_i + Bx_{i-1})$$

End

Stage III

$$x_{\log n} = L^{-1} f_{\log n}$$

B is upper triangular and A is (QxQ).

For optimal performance, Q should be choosen as large as possible. This keeps the algorithm in Stage I for the longest time. On the other hand, Stage I becomes inefficient when L is no longer reasonably full (Q > m) and when the individual calculations in Stage I require more processors than are available (p < Q*Q).

Stage II is just a block version of the classical column sweep algorithm which was discussed in the previous section.

To summarize the algorithm description, find values for L_i, where L is a $(2^i x 2^i)$ matrix, until the operations become

inefficient, then switch to multiple column sweeping to complete the solution.

The next section will discuss translating the algorithm into a description involving control structures.

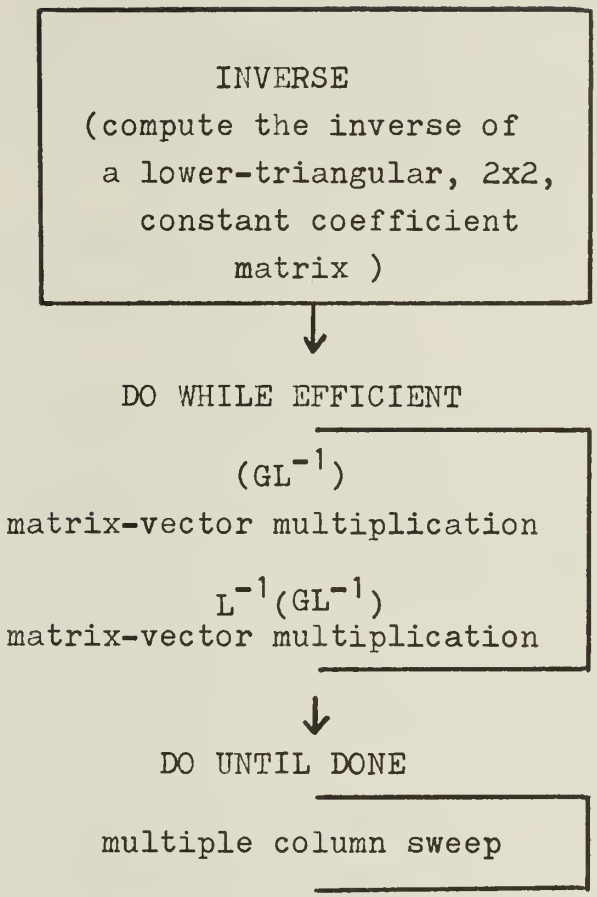
3.3.3 Implementing the Algorithm

Certain functions are used quite frequently by the algorithms discussed in this thesis. To avoid duplication and reduce unneeded complexity, MVMULT, CLSWP (both discussed earlier) and VVADD (vector-vector add) will be considered macros when defining the CCLGL algorithm. One final macro, BINVERS will also be used. This macro will refer to the calculation of the inverse of a banded constant coefficient 2x2 matrix, as discussed in the previous section. For completeness, INVERSE will invert a general 2x2 matrix.

Due to the definitions of the above macro's and the matrix representation of the CCLGL algorithm, a pictorial representation of the algorithm will be on a macro level (see fig. 3-12).

Since all control structures used in this algorithm have already been defined in the above macros, no new

structures need be added. The algorithm can be directly read from fig. 3-12 and is shown in fig. 3-13. This algorithm thus illustrates the ease of adding additional algorithms once a set of control structures and macros has been defined.


INVERSE
(compute the inverse of
a lower-triangular, 2x2,
constant coefficient
matrix)
DO WHILE EFFICIENT
(GL^{-1})
matrix-vector multiplication
$L^{-1}(GL^{-1})$
matrix-vector multiplication
DO UNTIL DONE
multiple column sweep

(fig. 3-12)

```
CCLGL:  Proc (N, M, P, Tptr) ;
     call DOLOOP (1) ;
          Q = COMPUTEQ (N, M, P) ;  /* finds 'best' time to
                                       switch to column sweep */
          call BINVERS (P, Pointer) ; /* inverse of 2x2 matrix */
          do i = 2 to CLOG2 (N) ;
               ibyi = ibyi * 2 ; /* double block size for
                                    every iteration */
               if ibyi > Q
               then do ;   /* finish with column sweep */
                    call CLSWP (Q, N, M, P, Pointer) ;
                    goto LEAVE ;
               end ;
               else if ibyi <= M
                    then do ;
                         call MVMULT (ibyi, ibyi, P, Pointer) ;
                         call MVMULT (ibyi, ibyi, P, Pointer) ;
                    end ;
                    else do ;
                         call MVMULT (M, M, P, Pointer) ;
                         call MVMULT (ibyi, M, P, Pointer) ;
                    end ;
               end ;  /* ends Do Loop */
   LEAVE: call MVMULT (N, N, P, Pointer) ; /* final results */
     call ENDO (Tptr) ;
end CCLGL ;
```

(fig. 3-13)

CHAPTER 4

STATISTICS

4.1 Introduction

In the previous chapters we presented the concept of control structures as a useful tool for analyzing algorithms for parallel machines. In this chapter we will look at several algorithms in detail. The emphasis of this study will be on the performance of the algorithms with respect to control structure usage.

For this chapter we will be looking at four algorithms: CCLGL (mentioned above), and ALGA, ALGB, and ALGC (see [6]). CCLGL is strictly for constant coefficient recurrences, the other three are for banded systems. ALGA is designed for P <= n, ALGB for P = O(n*m/2) and ALGC for P = O(m*m*n).

The set of resource times used throughout this chapter (except for section 4.5) will be unit times with the following values. For memory access, memory store, and both alignment in and alignment out, the time required for these resources is one time unit. For the processor steps, addition and subtraction and multiplication, we will use a

resource time of two time units. For division, three time units will be used, and for a unary minus, one time unit will be the resource time. Finally, we will use one time unit for a null processor step.

This set of times was chosen to represent a relatively typical set of resource times. Processor steps will usually take longer than data transfer steps. Also, division is a very time consuming processor step while both unary minus and a null processor operation are 'quick' processor steps.

No options, such as the slide option or the turning on of sneak paths, will be used without specifically mentioning this fact.

4.2 Occurences of Control Structures in Algorithms

4.2.1 Introduction

For these sections we will be interested in the frequency of use of selected control structures. We will look at which structures are used frequently, and how this affects the final times for these algorithms. First we will look at the algorithms separately, and then we will combine these results and look at the significance for recurrence algorithms in general.

For the analysis used in this section, we will be varying the values of three parameters. These parameters are n,m, and p. Here, p, as you will remember, refers to the number of processors available. In order to limit the number of runs needed, we have selected six representive values of p: 8,16,32,64,256,1024. These values have been chosen to represent a large cross section of processor values, ranging from a small number of processors (8), to a reasonably large number of available processors (1024).

The parameter n refers to the size of the recurrence. Once again we will restrict the number of values studied to a representive sample. Here we will use the following values: 10,20,50,100,500. This sample covers small recurrences (10,20,etc.) as well as an example of a large system (500) (Note: we are not assuming that N is a power of two).

The final parameter considered will be m, the bandwidth of the recurrence. Since most recurrences in practice will have small bandwidths, thus we have concentrated our efforts on small bandwidths, that is, m = 1,2,3. For a representation of large bandwidth recurrences, m = SQRT(N) and m = n - 1 have been choosen.

4.2.2 CCLGL

For small p (8), over 60% (approaches 99% for large n) of the processor time is spent during the last step of the algorithm. That is, most of the time is spent in the last matrix vector product, where the matrix is (NxN). While for p < O(n*n), folding takes place, for p > n the proportional time spent on this last step decreases dramatically. This speed up associated with p > n stems from the fact that with this many processors available, at least one row of the matrix can be done at a time (see section 3.3).

Two interesting exceptions to the above analysis can be found. For small p, significantly less time (proportionally) is spent in this last matrix product for systems having large bandwidths. This is not due to a reduction in the time spent on the last step, but rather reflects the additional time spent on the other phases of the algorithm for large values of m.

The other exception follows from the same line of thought presented in the previous paragraph. This time we will look at the other end of the spectrum. For large p, a significant proportion of the total time for the algorithm is once again spent in the last matrix-vector product. This time thus reflects the efficiency of the algorithm as a whole, when a large number of processors are available. In fact, for the case n = 500, m = 499 (the largest system

studied), the total time for the entire algorithm excluding the last step, was less than the time spent executing that last step.

A major interest in this algorithm stems from the choice of Q, the switchover point to column sweeping. Currently, the determination of Q requires Q > m. For some set of parameters, this bound switches the algorithm to column sweeping earlier than is optimal.

One example of this is for p = 256, where the n = 100, m = 99 system can be executed is less time than the system where n = 100, m = 10 or where n = 100 and m = 3. In both of these cases, the algorithm switches to column sweeping, while for m = 99 the algorithm never does switch.

Similar cases arise for other values of p. In these cases the amount of processor time increased while the total time decreased, compared to the the systems where column sweeping was never needed. These facts indicate that switching to column sweeping should be held off if the algorithm can be completed soon after the switch is currently called for. An additional criteria for Q, to minimize this inefficiency, might be to consider q > CEIL(N/CONSTANT), where the value of the constant can be determined in further studies.

4.2.3 ALGA

ALGA as mentioned earlier is designed for p <= n. Since part of the algorithm is dependent of log (p), large values of p can sometimes cause the algorithm to take longer, for the same recurrence, than when fewer processors are available. However there are several structures ('6FAFA*-<' and '5FAP?A?S' to name two) whose execution count is proportional to the value (n*m/p). Thus, too few processors can also slow down this algorithm.

For small m, p = O(n) gives the best results since (n*m/p) then has a small value. For large values of m, the best results are obtained with p = O(2*n) since with this bound the two conflicting factors mentioned above are best balanced.

4.2.4 ALGB

ALGB is designed for p = O(n*m/2). When executing this algorithm, performance is very sensitive to this bound. Basically, the entire time for the algorithm must be multiplied by CEIL((N*M)/(2*P)). For p >= (n*m/p) the times for this algorithm remain consistantly small. However, if p < (n*m/p), the algorithm becomes extremely slow.

Ignoring the folding, an interesting situation occurs as the value of m increases. Due to one bound based on log (n/m), the algorithm actually speeds up for large m. This effect is only noticable if no folding takes place. If folding takes place due to increasing the bandwidth, then this becomes a second order effect.

4.2.5 ALGC

ALGC is extremely dependent on m. Times for m = 2 are roughly twice those for m = 1, and m = 3 times are approximately twice those of m = 2, etc.

ALGC can be described in two parts. Part one is the initialization of the system, and part two is the actual execution phase. Part one is null for m = 1, thus accounting for the good times associated with this bandwidth. For m = 2, folding takes place for p < n. For p > n, only a few simple instructions need be executed to initialize the system, thus justifying the small times for this bandwidth. For m > 2, folding takes place if p < (m*m*m). Since a relatively large number of statements must be executed in order to initialize systems with a large bandwidth, the fact that folding takes place whenever the number of processors is less than m*m*m adds up to quite a few statements being executed to initialize systems with

$m > 2$.

Stage two is a relatively simple set of executable structures (no more than three), with each structure executed a relatively small number of times - at least if there is no folding. Folding occurs in this stage for $p < (m*m*n)$, thus once again the algorithm is extremely sensitive to large m.

4.3 The Effect of Sneak Paths

Sneak paths exist to bypass the established order of memory - alignment - processor - alignment - memory. Two of the four algorithms studied show no significant change when sneak paths around all resources are allowed. CCLFL as an algorithm never uses a structure that allows for sneak paths, and thus the times for this algorithm do not change when sneak paths exist.

ALGC does exhibit some time savings when sneak paths are present. This algorithm does contain some structures having sneak paths, however, the time saved by allowing sneak paths to exist is normally less than one percent of the total time. This result is due to two factors. One, the number of times structures containing sneak paths are executed compared to the number of times structures without

sneak paths are executed is insignificant. Two, because of overlap within structures, many resources that are skipped, due to sneak paths, are normally not included in the total time calculation. Thus, when their resources are skipped, no execution time is saved.

ALGA has several structures with resources that can be skipped due to sneak paths. Approximately a 10% time savings is realized by allowing the required sneak paths to exist. Some of this savings is due to the lack of processor operations within the structures containing these sneak paths ('2FA?', '2A?3', etc.). If sneak paths and the slide overlap options (see the next section) are available, some of the time savings due to sneak paths would be absorbed due by the effects of the slide option.

ALGB can be analyzed along the same lines as ALGA. While, in this algorithm, many structures containing sneak paths could be absorbed with the slide option, one factor prevents this from being as important as it is for ALGA. For ALGB, a large number of the structures containing sneak paths are memory to memory transfers ('FA?P?A?S' and 'FAP?A?S') which will not be totally absorbed by the slide option as mentioned for ALGA.

Fig. 4-1 displays the results from comparing the

algorithms assuming first no sneak paths, and then assuming all sneak paths are available. The numbers shown represent the percentage of the total time for the algorithms with sneak paths, compared to the total time without sneak paths.

% = T(all sneak paths on) / T(no sneak paths on)

	CCLGL		ALGA		ALGB		ALGC	
	p=16	p=32	p=16	p=32	p=16	p=32	p=16	p=32
N=10								
M=1	1.00	1.00	.90	.90	.84	.84	1.00	1.00
M=2	1.00	1.00	.88	.89	.88	.88	.98	.97
M=3	1.00	1.00	.92	.93	.90	.90	.96	.93
M=9	1.00	1.00	.96	.94	.84	.85	.99	.98
N=20								
M=1	1.00	1.00	.91	.90	.84	.84	1.00	1.00
M=2	1.00	1.00	.89	.90	.87	.86	.99	.98
M=3	1.00	1.00	.92	.93	.88	.90	.98	.97
M=4	1.00	1.00	.92	.93	.88	.89	.99	.98
M=19	1.00	1.00	.98	.97	.84	.84	1.00	1.00

(fig. 4-1)

N=50 (fig. 4-1 continued)

M=1	1.00	1.00	.89	.91	.83	.84	1.00	1.00
M=2	1.00	1.00	.88	.90	.87	.87	.99	.99
M=3	1.00	1.00	.91	.93	.88	.88	.99	.99
M=7	1.00	1.00	.93	.95	.88	.89	.99	.99
M=49	1.00	1.00	.99	.99	.83	.83	1.00	1.00
N=100								
M=1	1.00	1.00	.88	.89	.83	.83	1.00	1.00
M=2	1.00	1.00	.88	.89	.87	.87	1.00	1.00
M=3	1.00	1.00	.90	.92	.88	.88	1.00	1.00
M=10	1.00	1.00	.96	.96	.90	.90	1.00	1.00
M=99	1.00	1.00	1.00	1.00	.83	.83	1.00	1.00
N=500								
M=1	1.00	1.00	.87	.88	.83	.83	1.00	1.00
M=2	1.00	1.00	.87	.88	.87	.87	1.00	1.00
M=3	1.00	1.00	.90	.90	.88	.88	1.00	1.00
M=22	1.00	1.00	.98	.98	.92	.92	1.00	1.00
M=499	1.00	1.00	1.00	1.00	.80	.80	1.00	1.00

4.4 The Slide Overlap Option

The slide overlap option is described in [1]. The effect of this option is to merge structures based on data dependencies. The two key phrases here are structures and data dependencies. The more structures the more places for structures to slide together. The more data dependencies present, the more restrictions on when two structures can slide together, and thus the smaller the time savings due to this option.

Algorithms ALGA and ALGB require relatively few structures (repeated several times). Because of this, very little times savings appear due to the slide option. CCLGL and ALGC both require more structures and thus some savings from merging structures becomes evident.

Actually for all of the algorithms, this option has little effect. The reasons behind this have not been completely explored, but consider the number of structures required for the algorithm requiring the most structures. Even in this case, the number of structures used is under one hundred. Considering that the maximum time savings from a slide is the time it takes to execute one iteration of a structure, the maximum potential time savings from the slide option is small.

Fig. 4-2 displays the results of compairing the algorithms with and without the slide option. The numbers represent the percentage of total time allowing the slide option, divided by total time without the slide option.

% = T(slide option) / T(butt option)

	CCLGL		ALGA		ALGB		ALGC	
	p=16	p=32	p=16	p=32	p=16	p=32	p=16	p=32
N=10								
M=1	1.00	.95	1.00	1.00	1.00	1.00	1.00	1.00
M=2	1.00	.96	1.00	1.00	1.00	1.00	.89	.90
M=3	1.00	.96	1.00	1.00	1.00	1.00	.94	.93
M=9	1.00	.94	1.00	1.00	1.00	1.00	.99	.99
N=20								
M=1	1.00	.99	1.00	1.00	1.00	1.00	1.00	1.00
M=2	1.00	.99	1.00	1.00	1.00	1.00	.92	.89
M=3	1.00	.99	1.00	1.00	1.00	1.00	.96	.95
M=4	1.00	.99	1.00	1.00	1.00	1.00	.97	.95
M=19	1.00	.98	1.00	1.00	1.00	1.00	1.00	1.00

(fig. 4-2)

N=50 (fig. 4-2 continued)

M=1	1.00	.97	1.00	1.00	1.00	1.00	1.00	1.00
M=2	1.00	.98	1.00	1.00	1.00	1.00	.96	.93
M=3	1.00	.98	1.00	1.00	1.00	1.00	.98	.97
M=7	1.00	.95	1.00	1.00	1.00	1.00	1.00	.99
M=49	1.00	.99	1.00	1.00	1.00	1.00	1.00	.99

N=100

M=1	1.00	.98	1.00	1.00	1.00	1.00	1.00	1.00
M=2	1.00	.98	1.00	1.00	1.00	1.00	.98	.96
M=3	1.00	.99	1.00	1.00	1.00	1.00	.99	.98
M=10	1.00	.99	1.00	1.00	1.00	1.00	1.00	1.00
M=99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

N=500

M=1	1.00	.99	1.00	1.00	1.00	1.00	1.00	1.00
M=2	1.00	.99	1.00	1.00	1.00	1.00	.99	.99
M=3	1.00	.99	1.00	1.00	1.00	1.00	1.00	1.00
M=22	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
M=499	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

4.5 Comparing Algorithms with Various Resource Times

For this section we will be studying the effect of varying resource times. Time-A is the set of resource times mentioned in the previous sections. Time-B has every resource taking the same time, that is, one time unit. Time-C is a set of times with memory and alignment time equal to one, and all processor steps, except for the null processor step (one time unit), will require three time units to be executed.

By studying fig. 4-4 through fig. 4-7, the obvious observation is that as the processor time increases, the percentage of the total time spent executing processor operations increases. The only minor deviation from this is that although the processor time was increased linearly (from Time-B (1) to Time-A (app. 2) to Time-C (3))the percentage of processor time to total time increased at a slower rate. The reason for this can be traced to the fact that increasing the time of processor steps only increase potential overlap within structures. Since this study was done without the slide option, structures not containing a processor step will take the same time to execute no matter what the processor times are.

Among the algorithms, CCLGL has the highest percentage

of processor time vs. total time. This is due to two reasons. First, and most importantly, CCLGL is an algorithm designed for constant coefficient recurrences. By taking advantage of this fact, many memory and alignment steps may be skipped, leaving the same number of processor steps.

The second reason for the high concentration of processor steps is the efficiency of the overlap within the structures used while executing this algorithm. Since most of the execution time for this algorithm results from calls to MVMULT, and most of the structures in MVMULT are very close to processor bound, the above mentioned result is expected.

An interesting trend is noticed as the bandwidth increase for a fixed recurrence size (N). Generally, as the bandwidth increases, the percentage of processor time increases. This is probably due to the increase in the number of times each structure is executed.

Since overlap within a structure is optimal, the more time spent executing structures with significant overlap, the closer the total and processor times will coincide. One final observtion can be made by observing the trend associated with Time-C (processor time of three time units). Here, as the number of operations increases, the percentage

of time spent with the processor active increases and eventually approaches 100%. At this point, the possibility of an algorithm becoming processor bound exists. To avoid this, there should not be too large of a disparity between processor times and data transfer times.

The numbers shown in the following figures are the percentage of processor time vs. the total time for the algorithm.

COMPARING PROCESSOR TIME VS. TOTAL TIME

for VARIOUS RESOURCE TIMES (CCLGL)

	TIME-A		TIME-B		TIME-C	
	p=16	p=32	p=16	p=32	p=16	p=32
N=10						
M=1	.56	.53	.42	.38	.65	.60
M=2	.58	.54	.43	.39	.66	.62
M=3	.58	.55	.44	.41	.66	.62
M=9	.64	.54	.44	.41	.70	.60
N=20						
M=1	.67	.66	.42	.51	.73	.73
M=2	.67	.66	.44	.51	.74	.73
M=3	.67	.66	.45	.52	.73	.72
M=4	.67	.66	.45	.52	.73	.72
M=19	.75	.72	.55	.60	.86	.76

(fig. 4-3)

N=50 (fig. 4-3 continued)

M=1	.79	.77	.53	.52	.84	.81
M=2	.79	.76	.53	.52	.84	.81
M=3	.79	.76	.53	.53	.83	.80
M=7	.86	.75	.61	.54	.88	.79
M=49	.79	.80	.51	.58	.96	.91
N=100						
M=1	.86	.86	.57	.59	.90	.89
M=2	.86	.86	.57	.58	.90	.89
M=3	.86	.86	.58	.59	.89	.89
M=10	.91	.95	.67	.64	.96	.92
M=99	.79	.80	.50	.52	.99	.97
N=500						
M=1	.96	.96	.64	.64	.97	.97
M=2	.96	.96	.64	.64	.97	.97
M=3	.96	.96	.64	.64	.97	.97
M=22	.80	.99	.50	.68	1.00	.99
M=499	.80	.80	.50	.51	1.00	1.00

COMPARING PROCESSOR TIME VS. TOTAL TIME

for VARIOUS RESOURCE TIMES (ALGA)

	TIME-A		TIME-B		TIME-C	
	p=16	p=32	p=16	p=32	p=16	p=32
N=10						
M=1	.36	.36	.24	.24	.47	.47
M=2	.37	.36	.24	.24	.49	.50
M=3	.44	.44	.27	.27	.60	.62
M=9	.68	.67	.49	.41	.75	.78
N=20						
M=1	.37	.36	.24	.24	.48	.47
M=2	.41	.37	.26	.24	.52	.50
M=3	.48	.45	.29	.27	.62	.61
M=4	.54	.50	.34	.31	.66	.65
M=19	.83	.81	.67	.65	.87	.86

(fig. 4-4)

(fig. 4-4 continued)

N=50						
M=1	.44	.37	.28	.24	.56	.48
M=2	.45	.41	.28	.26	.55	.54
M=3	.53	.48	.33	.29	.64	.64
M=7	.71	.67	.49	.43	.79	.79
M=49	.91	.93	.82	.84	.94	.95
N=100						
M=1	.51	.43	.32	.37	.65	.55
M=2	.48	.44	.29	.27	.57	.55
M=3	.56	.52	.35	.32	.65	.65
M=10	.79	.78	.61	.56	.84	.84
M=99	.94	.95	.88	.90	.96	.97
N=500						
M=1	.69	.60	.43	.38	.87	.76
M=2	.51	.50	.30	.30	.58	.58
M=3	.59	.58	.38	.37	.67	.66
M=22	.88	.89	.75	.77	.91	.92
M=499	.36	.38	.37	.36	1.00	1.00

COMPARING PROCESSOR TIME VS. TOTAL TIME

for VARIOUS RESOURCE TIMES (ALGB)

	TIME-A		TIME-B		TIME-C	
	p=16	p=32	p=16	p=32	p=16	p=32
N=10						
M=1	.32	.32	.22	.22	.40	.40
M=2	.43	.43	.26	.26	.48	.48
M=3	.51	.51	.32	.32	.56	.56
M=9	.77	.74	.51	.50	.92	.88
N=20						
M=1	.32	.32	.22	.22	.40	.40
M=2	.46	.43	.29	.27	.53	.49
M=3	.56	.52	.36	.33	.63	.58
M=4	.64	.62	.42	.41	.73	.70
M=19	.82	.81	.54	.54	.98	.97

(fig. 4-5)

(fig. 4-5 continued)

N=50						
M=1	.34	.32	.23	.22	.42	.40
M=2	.49	.46	.30	.29	.56	.52
M=3	.58	.57	.38	.37	.66	.64
M=7	.75	.75	.52	.52	.85	.84
M=49	.83	.83	.55	.55	.99	.99
N=100						
M=1	.36	.36	.24	.23	.44	.41
M=2	.50	.49	.31	.30	.57	.55
M=3	.59	.58	.38	.37	.66	.65
M=10	.79	.79	.57	.57	.88	.88
M=99	.83	.83	.55	.55	1.00	1.00
N=500						
M=1	.36	.36	.25	.24	.45	.45
M=2	.49	.49	.30	.30	.56	.56
M=3	.58	.58	.37	.37	.65	.65
M=22	.86	.86	.67	.67	.94	.94
M=499	.80	.80	.50	.50	1.00	1.00

for VARIOUS RESOURCE TIMES (ALGC)

	TIME-A		TIME-B		TIME-C	
	p=16	p=32	p=16	p=32	p=16	p=32
N=10						
M=1	.56	.53	.22	.22	.46	.46
M=2	.58	.54	.26	.26	.57	.55
M=3	.58	.55	.35	.29	.68	.58
M=9	.64	.54	.35	.33	.77	.71
N=20						
M=1	.67	.66	.21	.22	.58	.46
M=2	.67	.66	.30	.26	.66	.58
M=3	.67	.66	.43	.38	.81	.72
M=4	.67	.66	.46	.42	.86	.79
M=19	.75	.72	.39	.38	.86	.83

(fig. 4-6)

(fig. 4-6 continued)

N=50						
M=1	.79	.77	.21	.21	.60	.58
M=2	.79	.76	.38	.33	.80	.72
M=3	.79	.76	.49	.47	.90	.86
M=7	.86	.75	.57	.56	.94	.93
M=49	.79	.80	.44	.44	.91	.90
N=100						
M=1	.86	.86	.26	.21	.70	.60
M=2	.86	.86	.41	.38	.85	.80
M=3	.86	.86	.52	.50	.93	.90
M=10	.91	.95	.62	.62	.96	.95
M=99	.79	.80	.50	.45	.98	.93
N=500						
M=1	.96	.96	.31	.29	.85	.80
M=2	.96	.96	.44	.43	.91	.90
M=3	.96	.96	.53	.53	.95	.94
M=22	.80	.99	.63	.59	.93	.90
M=499	.80	.80	.49	.49	.96	.96

4.6 Comparing Actual and Theoretical Time Bounds

Several papers have mentioned theoretical bounds on the algorithms mentioned in this chapter. One such work, [6], gives the processor bounds for ALGC as:

$$t = \lceil \log_2 n \rceil (2 + \lceil \log_2 m \rceil) - \lceil \log_2 m \rceil (\lceil \log_2 m \rceil + 1)/2$$

Fig. 4-8 shows the theoretical processor time, the actual processor time, and the total time for ALGC. For this study, we have used the resource time parameters refered to in section 4.5 as Time-B. This set of resource times allows one time unit for the execution of every resource request.

The theoretical and actual processor times do not exactly coincide because of memory-memory transfers ('FAPAS'). Of course, the total time includes data transfers and thus is greater than either the theoretical or actual processor time.

We have restricted this study to recurrences that can be solved using ALGC without folding. In order to include as many different recurrences as possible, we used p = 1024 to avoid folding. The entries in fig. 4-8 are the actual times and not percentages as displayed in previous figures.

ALGC

P = 1024; p > m*m*n

	THEORETICAL PROCESSOR TIME	ACTUAL PROCESSOR TIME	TOTAL TIME
N=10			
M=1	8	8	36
M=2	11	14	53
M=3	13	16	60
M=4	13	16	60
M=7	14	18	75
N=20			
M=1	10	10	45
M=2	14	17	64
M=3	17	20	20
M=4	17	20	20
M=7	19	23	88

(fig. 4-8)

n=50			
M=1	12	12	54
M=2	17	20	75
M=3	21	28	96
M=4	21	24	84
N=100			
M=1	14	14	63
M=2	20	23	86
M=3	25	32	108
N=250			
M=1	16	16	72
M=2	23	26	97
N=1000			
M=1	20	20	90

(fig. 4-8 continued)

4.7 Future Studies

The studies presented in this chapter are indicitive of the studies and considerations made possible by the concept of control structures. These studies are not intended to be a definitive study of the four algorithms we discussed. Many future studies can be made given the flexibility of these structures.

The main area of interest illuminated by the studies done for this chapter is the effect of combining the premise behind the studies already made. We have already mentioned the potential interest in combining the sneak path and the slide overlap studies.

Of further note are the cases of varying resource times with one or both of the options just mentioned. With processor time much greater than memory or alignment time, the slide option could potentially reduce the total time for the algorithms to almost that of just considering the processor time. Making memory and alignment time large will increase the improtance of both the slide option and the presence or absence of sneak paths.

These and many other studies can be made to determine a 'best' set of resources to execute these structures.

CHAPTER 5

CONCLUSION

We have presented a concept for allocating resources at the machine instruction level. By allocating several resources as a single instruction, overlap between resources can be statically calculated at the time of defining a given instruction. This allows for optimal overlap within a control structure and minimizes the need for dynamic overlap calculations.

As we studied several recurrence algorithms, it became apparent that very few unique control structures are sufficient for describing the algorithms currently under consideration. Thus, a limited number of control structures/machine instructions are capable of handling the known recurrence algorithms.

The combination of the above two factors, limited number of control structures and their inherent efficiency, makes this concept of control structures potentially useful as a source of machine instructions for parallel machines.

We have shown the adaptability of this concept for handling current algorithms as well as potential future algorithms. We have also illustrated the ease by which algorithms can be described in terms of control structures and thus we illustrated the inherent simplicity of executing an algorithm composed of these structures. In addition to their usefulness as machine instructions, it was demonstrated that control structures make the analysis of algorithms easy to conceptualize. Thus control structures are well suited for studies to determine the best computer architecture to maximize the speed and efficiency of executing recurrence algorithms on parallel machines.

REFERENCES

[1] J. C. Wawrzynek,
"Scheduling and Simulation of Computation Graphs for Resource Computers,"
Master's Thesis, University of Illinois at Urbana, May, 1979.

[2] D. D. Gajski,
"An Algorithm for Solving Linear Recurrence Systems on Parallel and Pipelined Machines,"
Dept. of Computer Science, University of Illinois at Urbana, unpublished memo, November, 1978.

[3] R. K. Montoye,
"Parallel and Pipeline Solution of First Order Difference Problems by Odd Equation Elimination,"
Dept. of Computer Science, University of Illinois at Urbana, unpublished memo, August, 1978.

[4] D. H. Lawrie,
"Recurrence Implementation Notes - I, 'Constant Coefficient Algorithms'," Dept. of Computer Science, University of Illinois at Urbana, unpublished memo, File 168, August, 1977.

[5] S. C. Chen, D. J. Kuck, and A. H. Sameh, "Practical Parallel Band Triangular System Solvers," ACM transactions on Mathmatical Software, vol.4, no.3, Sept. 1978, pp.270-277.

[6] K. Y. Wen, "Interprocessor Connections - Capabilities, Exploitation and Effectiveness," PhD. Thesis, University of Illinois at Urbana, UIUCDCS-R-76-830, October, 1976.

APPENDIX A

Appendix A presents the set of control structures needed to execute the algorithms discussed in this thesis. First we show the control structures to graphically illustrate the relationship between the resources contained within each structure. To avoid duplication, we use here the notation introduced in the section that defined the canonical set. That is, a generalized processor step, P, is used, and sneak paths are ignored. Simply for ease in locating structures, the structures have been divided, as per Chapter 2, into a canonical set and a set of additional structures. The rest of Appendix A lists the actual structures used and indicates which sturctures are needed for which algorithms.

CANONICAL SET

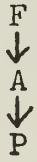
2FA

2AS

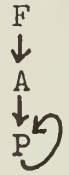
2AP

2AP<

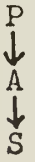
3FAP



3FAP<

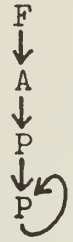


3PAS



4FAPP

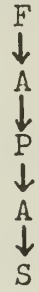
4FAPP<



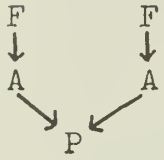
ADDITIONAL STRUCTURES

4APAS

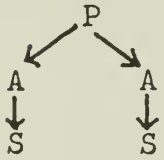
5FAPAS



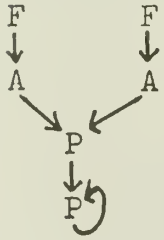
5FAFAP



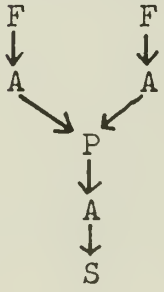
5PASAS



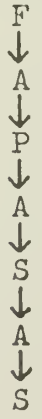
6FAFAPP<



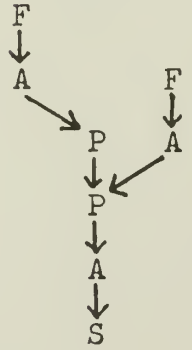
7FAFAPAS



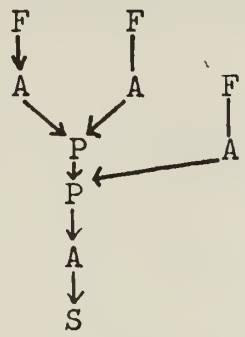
7FAPASAS



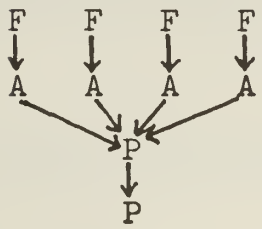
8FAPFAPAS



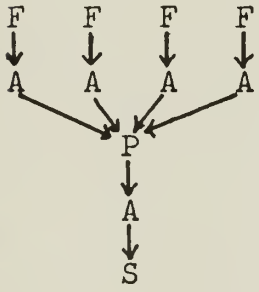
10FAFAPFAPAS



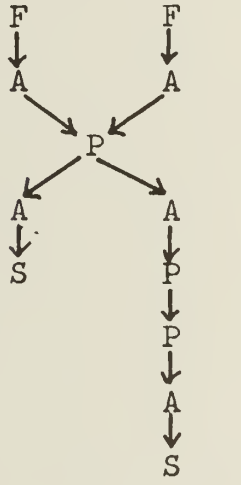
10FAFAFAFAPP<



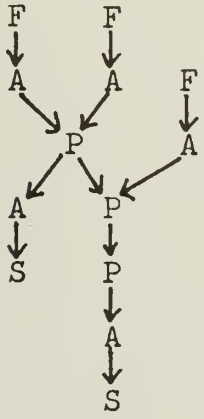
11FAFAFAFAPAS



12FAFAPASAPPAS



13FAFAPASFAPPAS



CONTROL STRUCTURES AND WHERE THEY ARE USED

The Structures	ALGA	ALGB	ALGC	CCLGL	MVMULT
2AS	x	x	x	x	x
2FA	x	x			x
2A+<			x		
2A+<				x	x
2A*				x	
3FA+			x		
3U-AS				x	
4FA*+<				x	x
4A+AS			x		
5FA+AS	x		x		
5FAFA*			x		
5FAPAS	x	x	x		
5FA*AS				x	
5FAU-AS				x	
5U-ASAS	x				

5FA-AS			x		
6FAFA*-<	x	x			
6FAFA*+<				x	x
7FAFA*A+<			x		
7FAFA+AS			x	x	
7FAFA*AS			x		
7FAFA/AS				x	
7FAPASAS		x			
8FA*FA+AS				x	x
10FAFAFAFA*-<	x				
10FAFAFAFA*+<	x				
10FAFA*FA-AS	x				
10FAFA*FA+AS		x		x	x
11FAFAFAFA*AS			x		
12FAFA/ASA*U-AS				x	x
13FAFA/ASFA*U-AS				x	x

APPENDIX B

Appendix B presents the algorithms discussed in this thesis. CCLGL and MVMULT are discussed in Chapter 2 and thus will not be repeated here. This appendix will contain the three algorithms refered to as ALGA, ALGB, ALGC. For a more detailed description of these algorithms, see [6].

The algorithms will be presented in a psydo-code version. The emphasis of this appendix is to show the conditions for the use of control structures. MAP signifies a call to FAPGEN ([1]) and designates a control structure and its repetition factor. DEPEND flags data dependencies (see Chapter 2). CEIL is the ceiling function. DOLOOP and ENDO are used to form a loop around structures.

```
ALGA: Proc (n,m,p,pointer) ;
call DOLOOP (1) ;
   bound = CEIL (n*m/p) - 1;
   call MAP ('10FAFA*FA-AS',M) ;
   call MAP ('5FAP?A?S',M) ;
   call DEPEND (5,1) ;
   if m = 1
   then call MAP ('10FAFA*FA?-A?S',BOUND) ;
   else do ;
      call DOLOOP (bound) ;
         call MAP ('2FA?',1) ;
         call DEPEND (2,6) ;
         call MAP ('6FAFA*-<',M) ;
         call DEPEND (6,1) ;
         call MAP ('2A?S',1) ;
      call ENDO (pointer) ;
   call TRANSPOSE (m,p,pointer) ; /* an MxM matrix */
   call MAP ('5FA?P?A?S',1) ;
   bound2 = LOG  (P/(M)  - 1 ;
   call DEPEND (5,1) ;
   call DOLOOP (bound2) ;
      /* partition memories */
      bound3 = CEIL (m/2) ;
      call MAP ('2FA',1) ;
      call DEPEND (10,1) ;
      call MAP ('10FAFAFAFA*-<'BOUND3) ;
```

```
   call DEPEND (10,1) ;
   call MAP ('5FA+A?S',1) ;
   call DEPEND (5,1) ;
   call DOLOOP (bound3) ;
      call MAP ('10FAFAFAFA*+<',M) ;
      call DEPEND (10,1) ;
      call MAP ('5U-A?SA?S',1) ;
   call ENDO (pointer) ;
  call ENDO (pointer) ;
call DEPEND (5,1) ;
call DOLOOP (1) ;
   /* partition memories */
   call MAP ('2FA',1) ;
   call DEPEND (2,10) ;
   bound4 = CEIL (m/2) ;
   call MAP ('10FAFAFA*-<',BOUND4) ;
   call DEPEND (10,1) ;
   call MAP ('5FA+A?S',1) ;
call ENDO (pointer) ; /* ends stage 3 */
   /* partition memory */
if remote.term then ; /* no need for last step */
else do ;
   bound5 = CEIL (n/p) -1 ;
   call DEPEND (5,1) ;
   if m = 1
   then call MAP ('10FAFA*FA?-A?S',BOUND5) ;
```

```
      else do ;
         call DOLOOP (bound5) ;
            call TRANSPOSE (m,p,pointer) ;
            call DEPEND (5,1) ;
            call MAP (2FA?',1) ;
            call DEPEND (2,1) ;
            call MAP ('6FAFA*-<',M) ;
            call DEPEND (6,1) ;
            call MAP ('2A?S',1) ;
         call ENDO (pointer) ; /* ends stage 4 */
      end ;
   end ;
call ENDO (pointer) ; /* ends algorithm */
end ALGA ;
```

```
ALGB: Proc (n,m,p,pointer) ;
fold = CEIL (n*m/(2*p)) ;
call DOLOOP (1) ;
   call DOLOOP (2) ; /* stage 1 */
      call110 ('2FA',1) ;
      call DEPEND (2,8) ;
      temp = 2 * m * fold ;
      call MAP ('10FAFA*FA?+A?S',TEMP) ;
      call DEPEND (10,1) ;
   call ENDO (pointer) ; /* ends initialization */
   fold = CEIL (m*n/(2*p) ;
   bound = LOG  (N/M)  - 1 ;
   temp = bound * fold ;
   call DOLOOP (temp) ;
      call MAP ('2FA?',1) ;
      call DEPEND (2,5) ;
      call MAP ('5FAFA*-<',M) ;
      call DEPEND (6,1) ;
      call MAP ('5FAP?A?S',1) ;
      call DEPEND (5,1) ;
      call MAP ('2AS',1) ;
      call DEPEND (2,1) ;
      call MAP ('6FAFA*-<',M) ;
      call DEPEND (6,1) ;
      call MAP ('5FAP?A?S',1) ;
      call DEPEND (5,1) ;
```

```
      call MAP ('2AS',1) ;
      call DEPEND (2,1) ;
   call ENDO (pointer) ;
   call DEPEND (7,1) ;
   call MAP ('2FA?',1) ;
   call DEPEND (2,5) ;
   if remote.term then ; /* can skip the rest */
   else do ;
      call DOLOOP (fold) ;
         call MAP ('6FAFA*-<',M) ;
         call DEPEND (6,1) ;
         call MAP ('7FAP?A?SAS',1) ;
      call ENDO (pointer) ;
   end ;
call ENDO (pointer) ; /* ends this algorithms */
end ALGB ;
```

```
ALGC: Proc (n,m,p,pointer) ;
call Doloop (1) ;
   if m = 1 then ; /* no initialization needed */
   else if m = 2 then do ; /* initialize */
      fold = CEIL (n/p) ;
      if fold > 1
      then do ;
         call MAP ('7FAFA*AS',FOLD) ;
         call DEPEND (5,1) ;
         call MAP ('3FA+<',FOLD) ;
         call DEPEND (3,1) ;
         call MAP ('2AS',1) ;
         call DEPEND (2,1) ;
      end ;
      else do ;
         call MAP ('7FAFA*A+<',FOLD) ;
         call DEPEND (7,1) ;
      end ;
      call MAP ('5FA-AS',FOLD) ;
      call DEPEND (5,1) ;
      temp = 2 * fold ;
      call MAP ('5FAP?A?S',TEMP) ;
      call DEPEND (5,1) ;
   end ; /* ends case where m = 2 */
   else do ; /* cases m > 2 */
      q = 2 ; /* this is basically ALGFULL (see [6]) */
```

```
   bound =  LOG  (M)  -2 ;
   do i = 0 to bound ;
      fold = CEIL (m*q*q*2/p) ;
      call MAP ('11FAFAFAFA*AS',FOLD) ;
      if i > 0
      then do ;
         fold = CEIL (m*m*q/p) * i ;
         call MAP ('7FAFA+AS',FOLD) ;
      end ;
   call DEPEND (7,1) ;
   fold= CEIL (m/p) ;
   call MAP ('7FAFA?+A?S',FOLD) ;
   call DEPEND (7,1) ;
   fold = CEIL (m*m/p) ;
   call MAP ('5FAP?A?S',FOLD) ;
   call DEPEND (5,1) ;
   q = q * 2 ; /* begin next iteration */
end ;
call MAP ('5FAP?A?S',1) ; /* an additional align */
call DEPEND (5,1) ;
fold = CEIL (m*m*m/p) ;
bound =  LOG (M)  ;
temp = bound * fold ;
if fold > 1
then do ;
   call MAP ('4A+AS',TEMP) ;
```

```
   call DEPEND (2,1) ;
end ;
else do ;
   call MAP ('2A+<',TEMP) ;
   call DEPEND (2,1) ;
end ;
fold = CEIL (m*m/p) ;
call MAP ('5FA?+A?S',FOLD) ;
call DEPEND (5,1) ;
end ; /* ends initialization step */
bound =  LOG (N/M) ;
call DOLOOP (bound) ;
   if remote.term /* no need for folds */
   then do ;
      call MAP ('5FAFA*',1) ;
      call DEPEND (5,1) ;
      temp =  LOG (M) ;
      if m = 1 then ;
      else do ;
         call MAP ('2A+<',TEMP) ;
         call DEPEND (2,1) ;
      end ;
      call MAP ('5FA-AS',1) ;
      call DEPEND (5,1) ;
   end ; /* ends remote term case */
   fold = CEIL (m*m*n/p) ;
```

```
      call MAP ('7FAFA*AS',FOLD) ;
      call DEPEND (5,1) ;
      temp = fold *  LOG (M)  ;
      if m = 1 then ;
      else do ;
         call MAP ('4A+AS',TEMP) ;
         call DEPEND (2,1) ;
      end ;
      fold = CEIL (m*m*n/(2*p)) ;
      call MAP ('5fa-as',fold) ;
      call DEPEND (5,1) ;
   end ; /* ends variable coefficient case */
call ENDO (pointer) ; /* ends stage 2 */
call ENDO (pointer) ; /* ends algorithm */
end ALGC ;
```

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-79-973	2.	3. Recipient's Accession No.

4. Title and Subtitle

Control Structures for Parallel Machines

5. Report Date
May, 1979

6.

7. Author(s)
Frank Dagen Panzica

8. Performing Organization Rept. No. UIUCDCS-R-79-973

9. Performing Organization Name and Address

University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.
US NSF MCS76-81686

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, D. C.

13. Type of Report & Period Covered
Master's Thesis

14.

15. Supplementary Notes

16. Abstracts

Until recently, algorithms for solving recurrence have been judged by counting processor steps. The purpose of this report is to explore and adapt these algorithms for a generalized parallel machine. Machine architecture parameters such as sneak paths, register usage and resource times will be considered, along with potential scheduling problems.

While considering these factors, the concept of control structures will be introduced. The potential of defining machine instructions to execute these control structures has been considered and will be mentioned throughout this report.

17. Key Words and Document Analysis. 17a. Descriptors

Control structures
Machine instructions
Recurrence algorithms

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement

Release Unlimited

19. Security Class (This Report) UNCLASSIFIED

20. Security Class (This Page UNCLASSIFIED

21. No. of Pages 120

22. Price